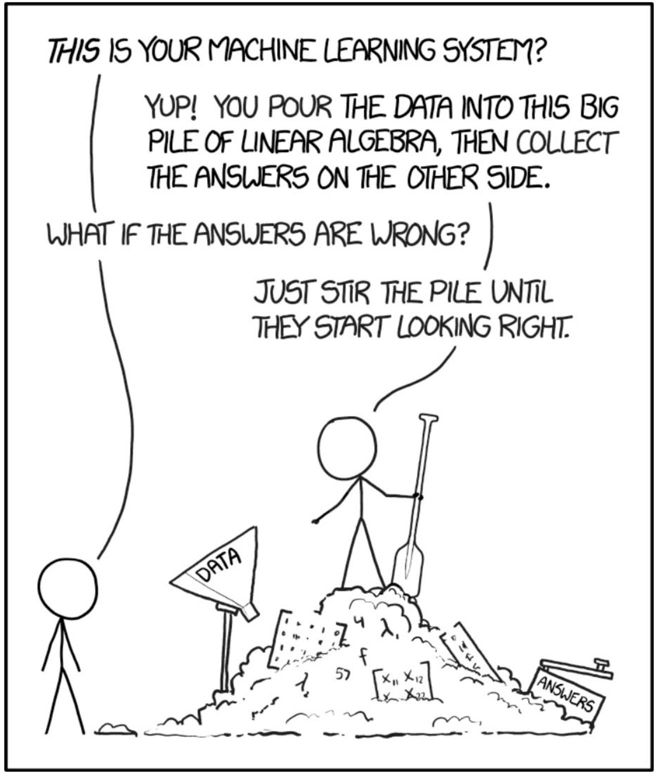
Accelerating Deep Learning*

CPEN 311



xkcd #1838

a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

*be very suspicious: computer architect on the loose talking about machine learning

A warp-speed intro to deep learning

- some philosophy
- multilayer perceptrons
- convolutional neural networks
- training?
- computational patterns
- accelerators

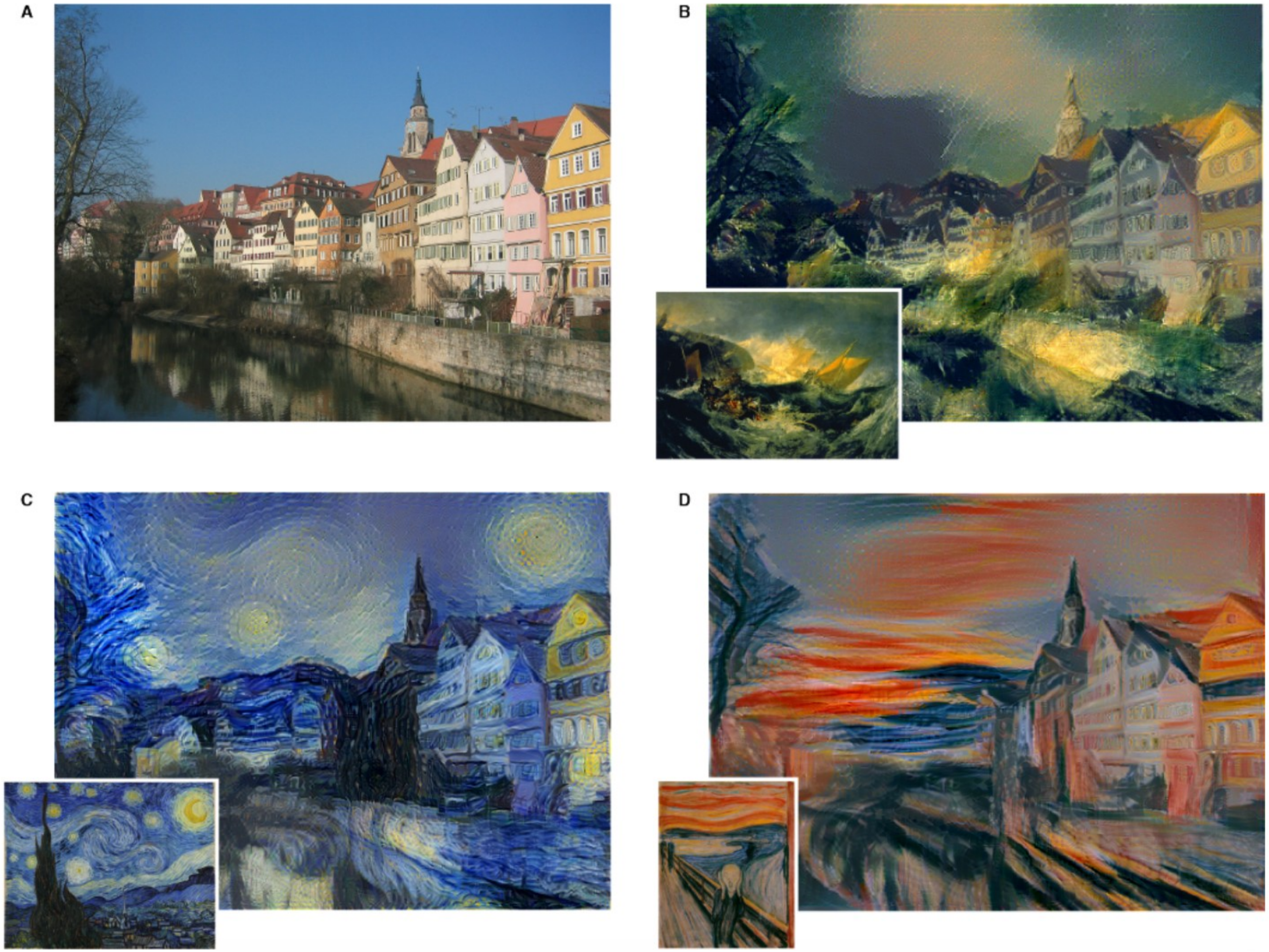
A warp-speed intro to deep learning

- some philosophy
- multilayer perceptrons
- convolutional neural networks
- training?
- **computational patterns**
- **accelerators**

you will build a very simple accelerator in the lab

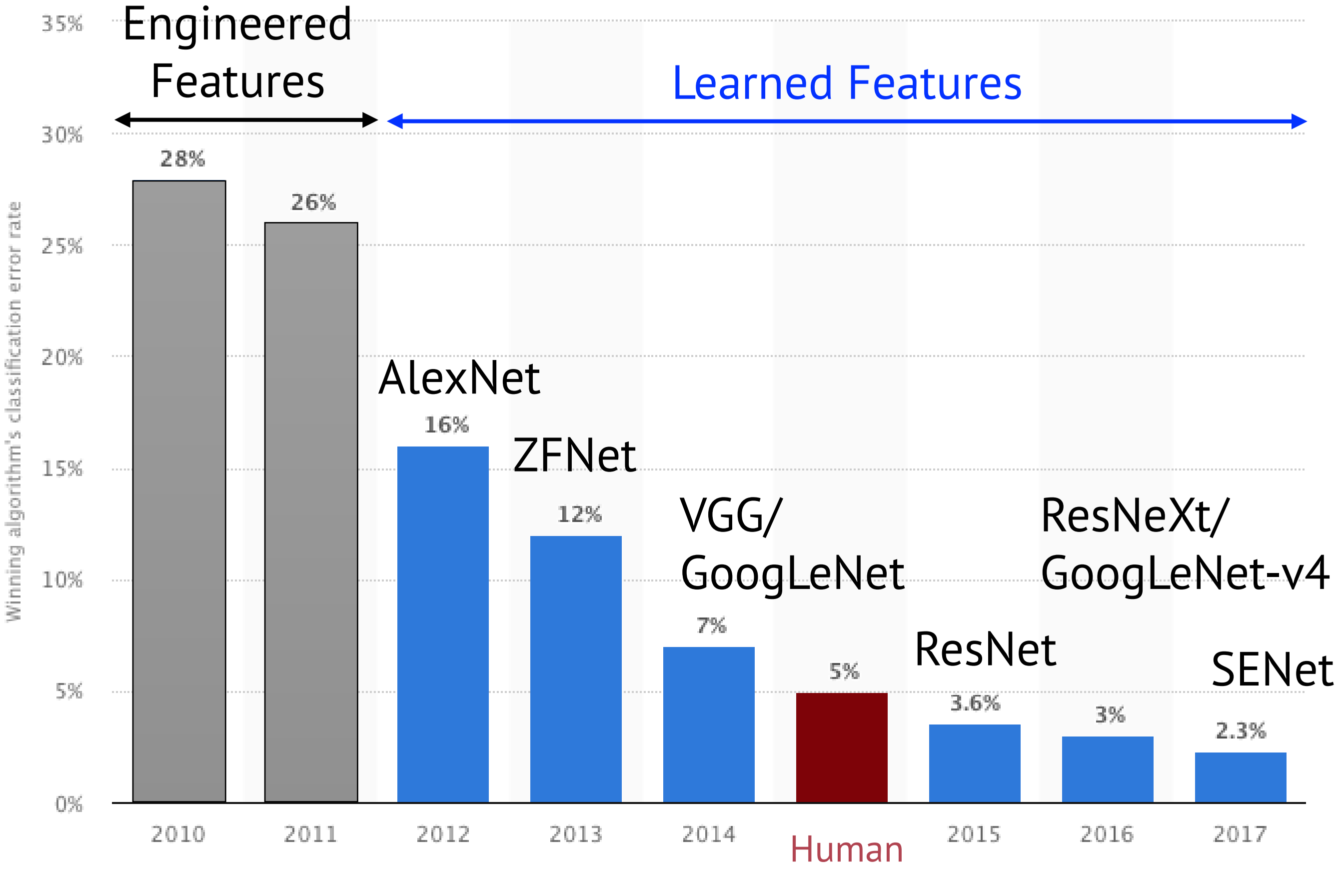
image: Tom Mihalek/AFP/Getty Images

image: Lee Jin-man/AP



Gatys+. A Neural Algorithm of Artistic Style. *arXiv:1508.06576*, 2015.

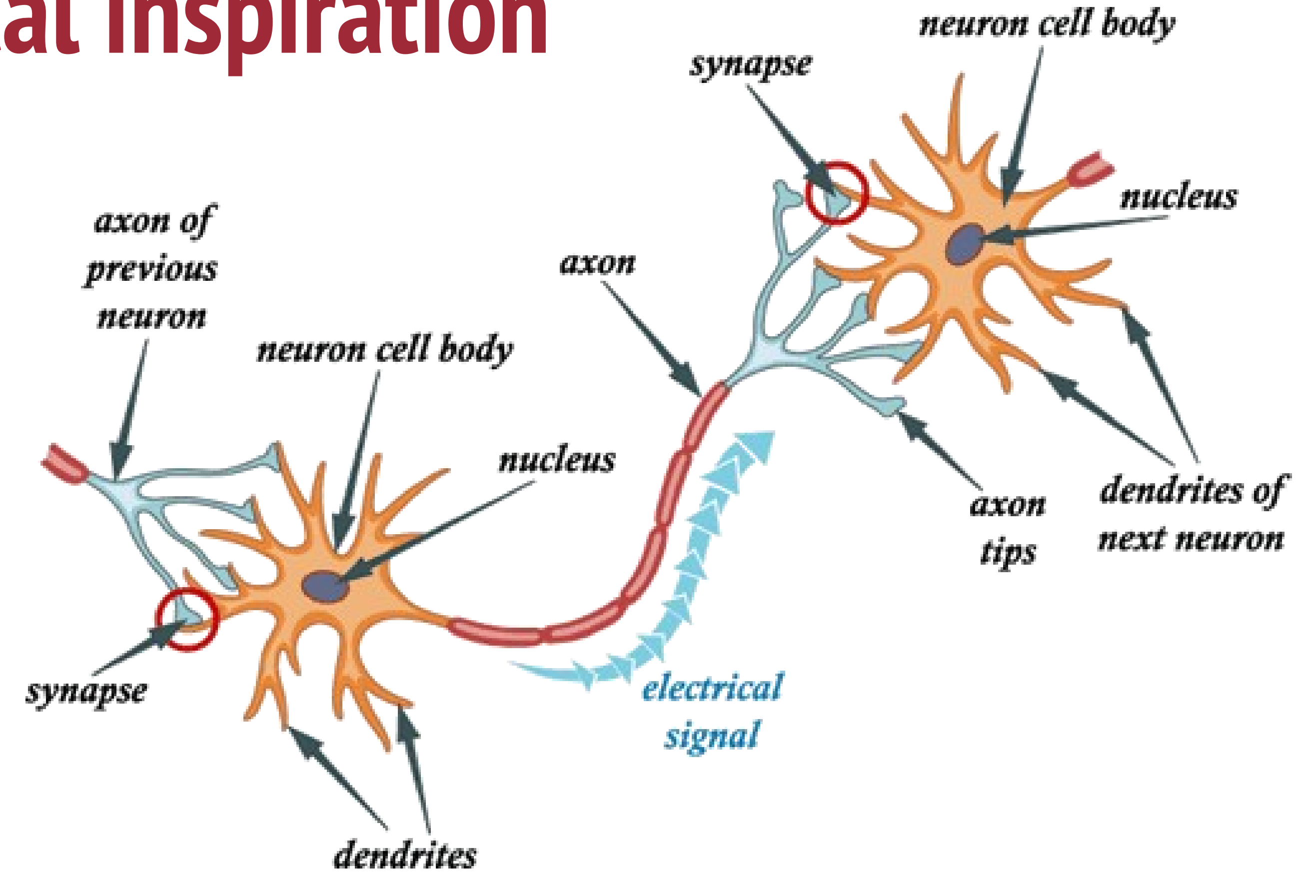
Deep Learning Revolution (2012+)



part 1:

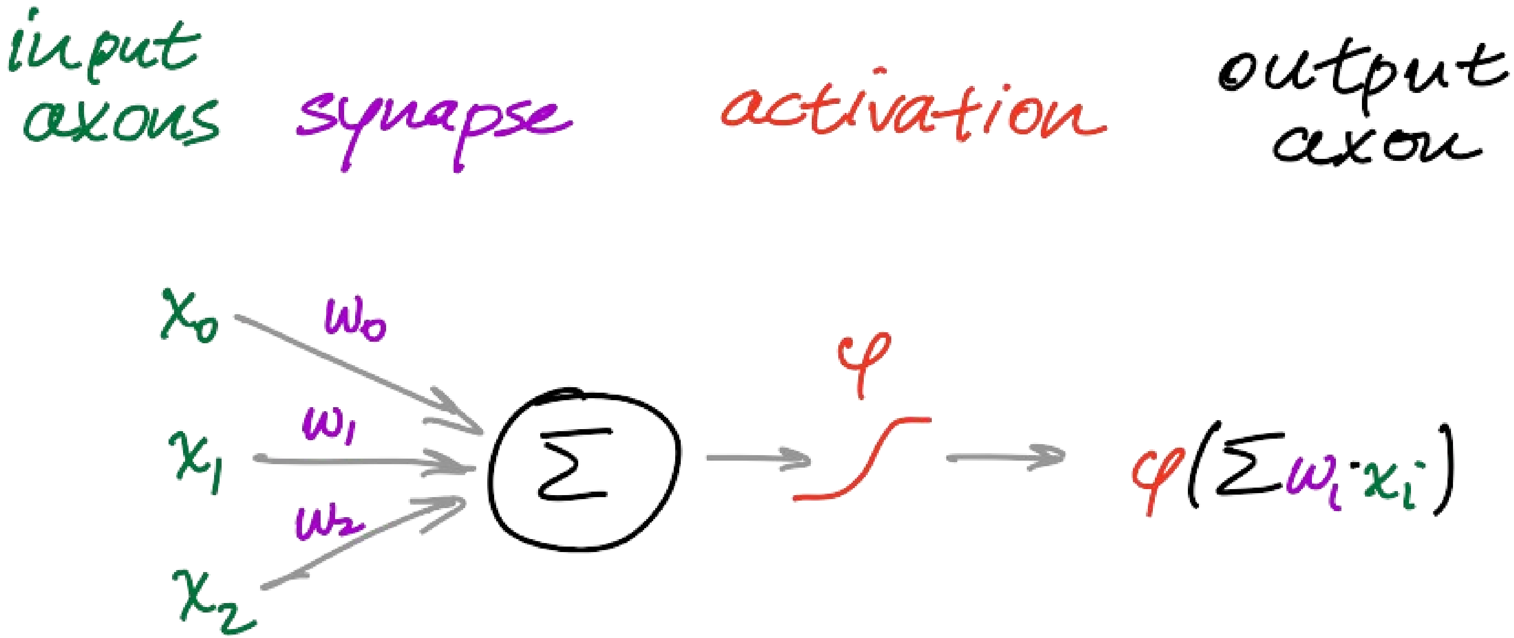
Artificial Neural Networks

Biological inspiration

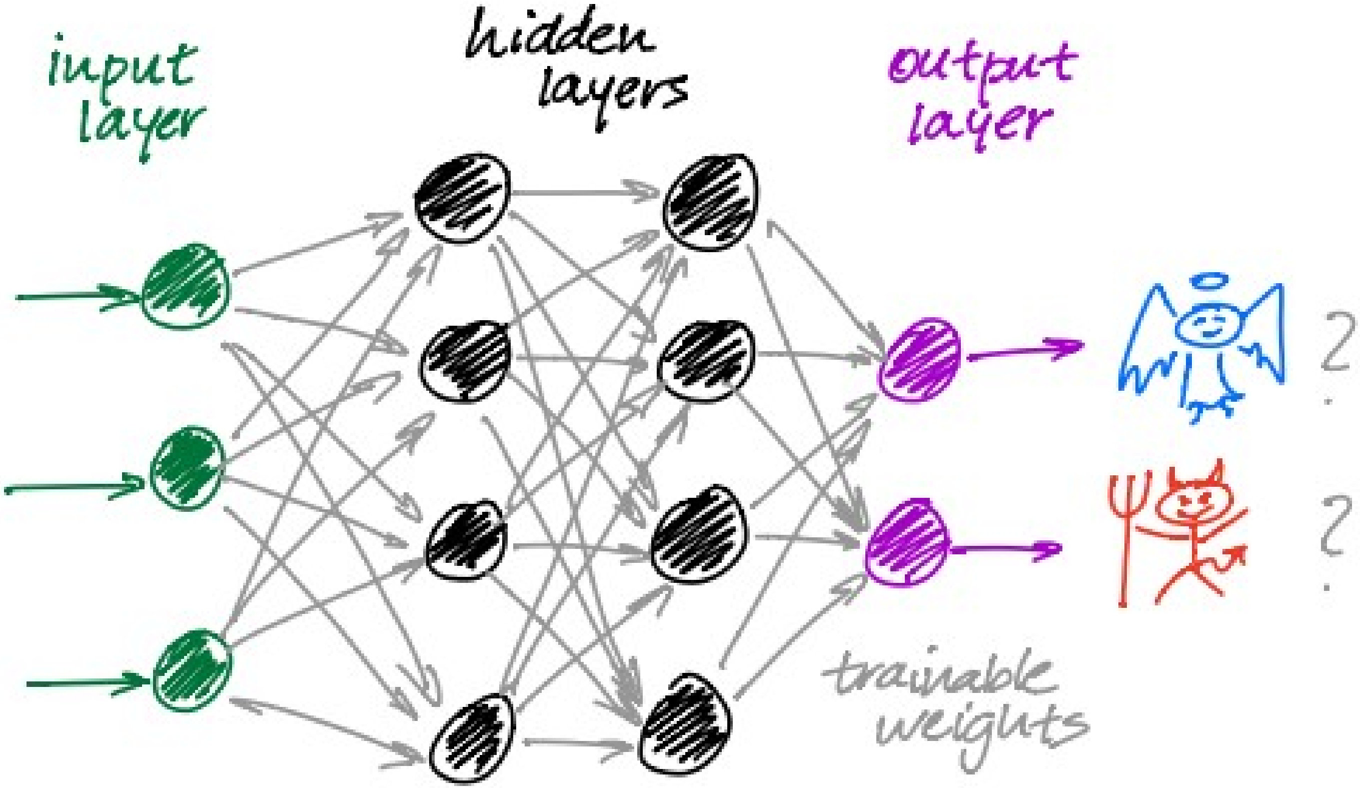


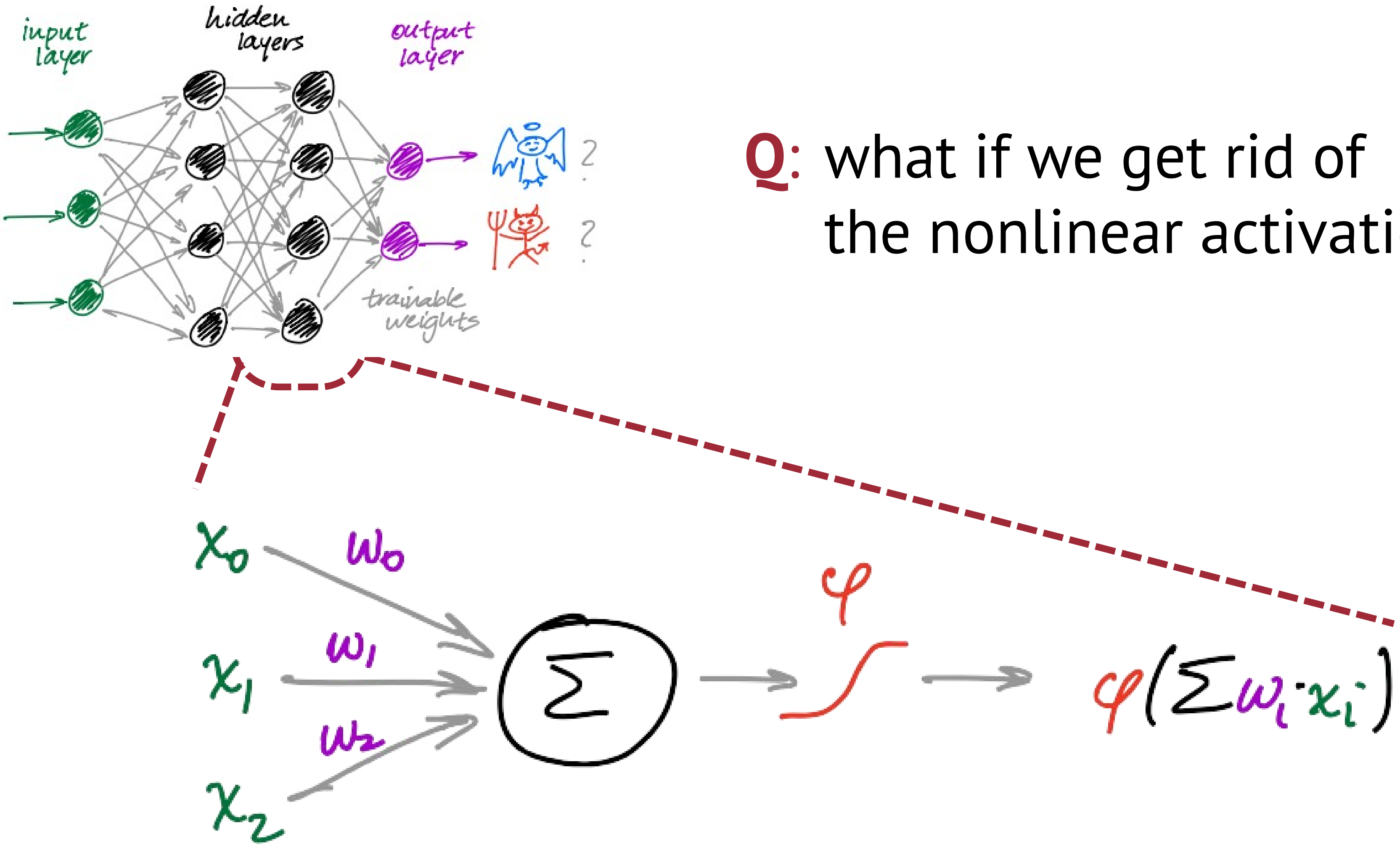
I couldn't find the correct attribution for this image :(

An artificial neuron model



Multilayer Perceptron (MLP)





Q: what if we get rid of the nonlinear activations?

Can represent fairly complex functions

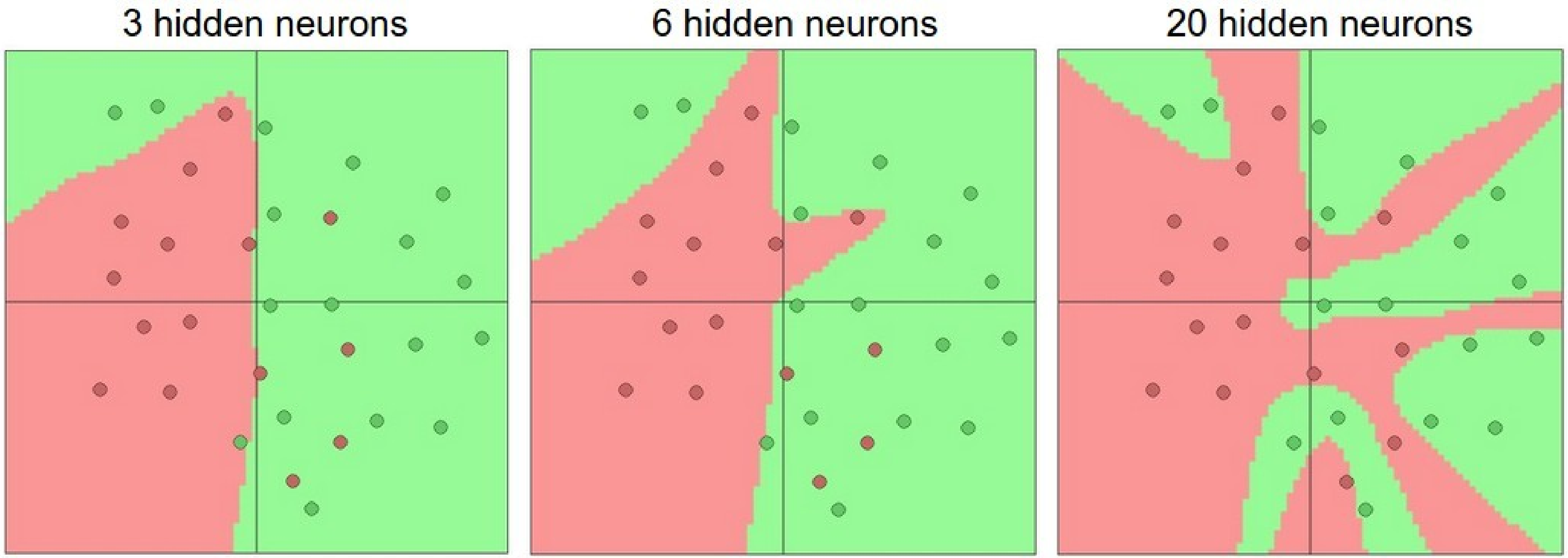
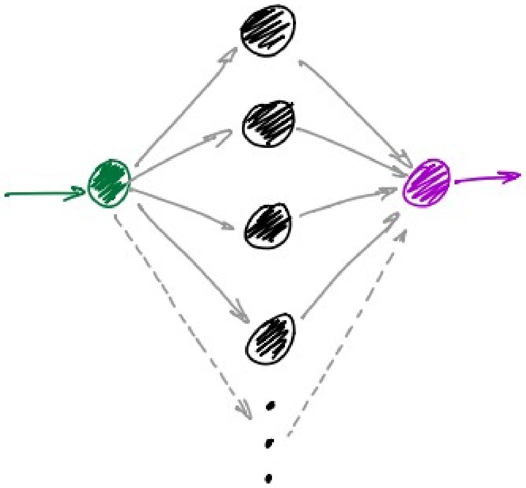
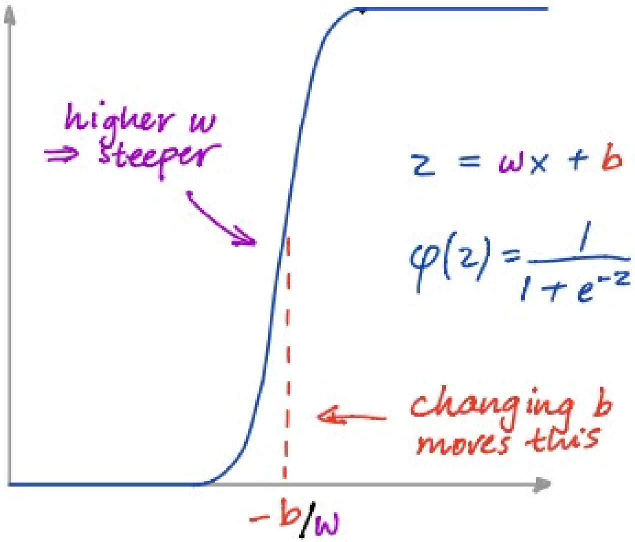
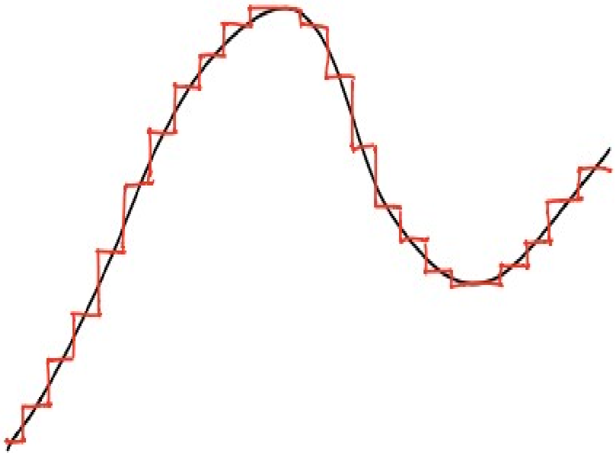


image: CS231n (Stanford)

Can approximate ALL continuous real fns*







but: can't learn hierarchy of concepts → training very hard

*Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Math. Control Signals Systems* 2:303, 1989.

part 2:
Convolutional Neural Nets

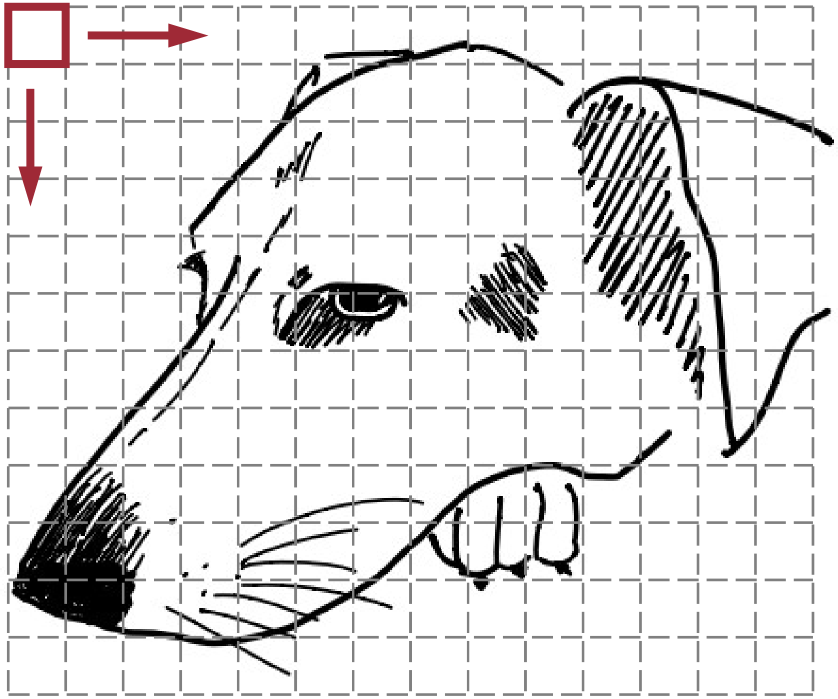
edges and corners everywhere and **key to recognition**

but an MLP needs to learn edge @ each location separately!

Convolutional layers

IDEA: convolutional filters

- learn $n \times n$ **filters** (e.g., 5×5) to detect, e.g., edges
- apply each **everywhere** across the image

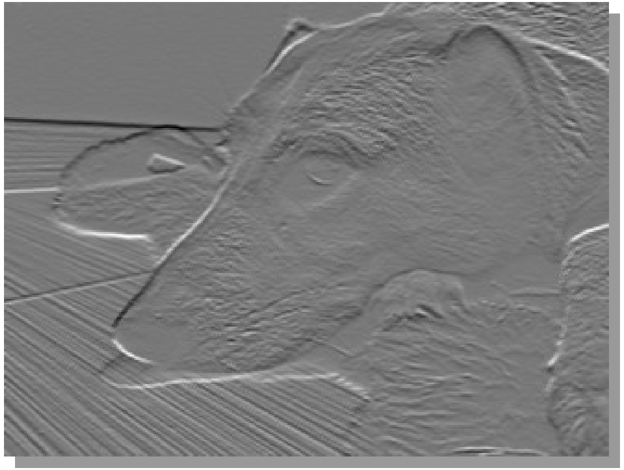


learns **features** that are **invariant to translation**

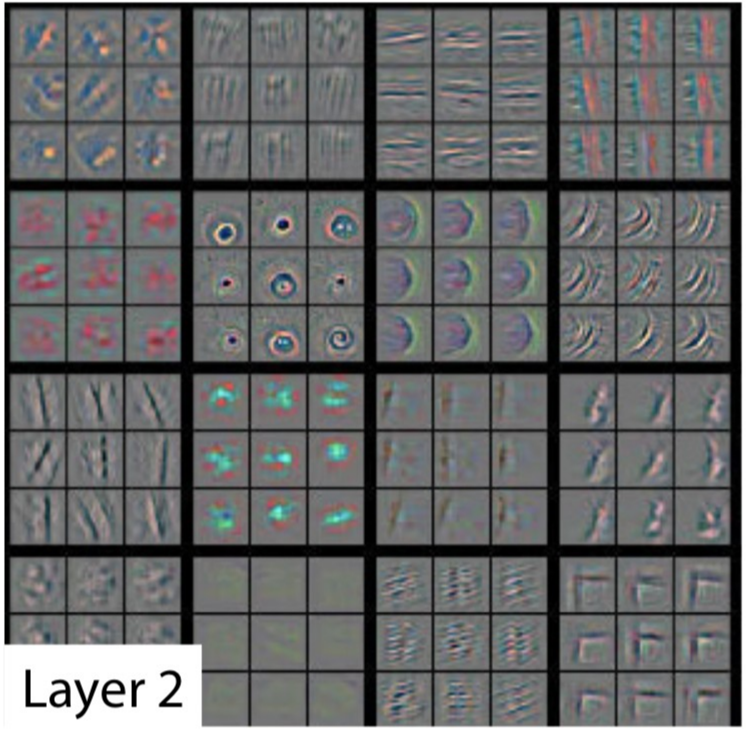
Convolution filters

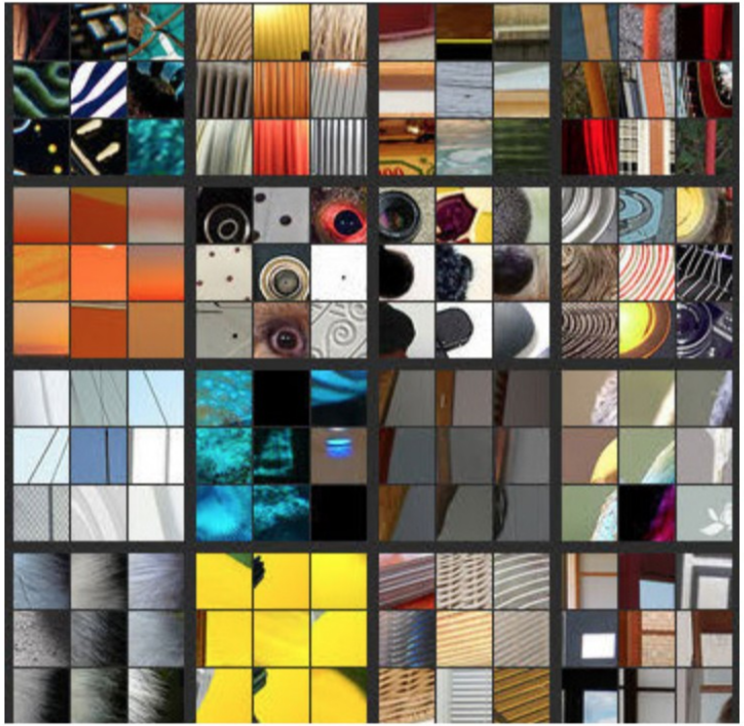
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} * \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = aA + bB + cC + dD + eE + fF + gG + hH + iI$$

$$* \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} =$$

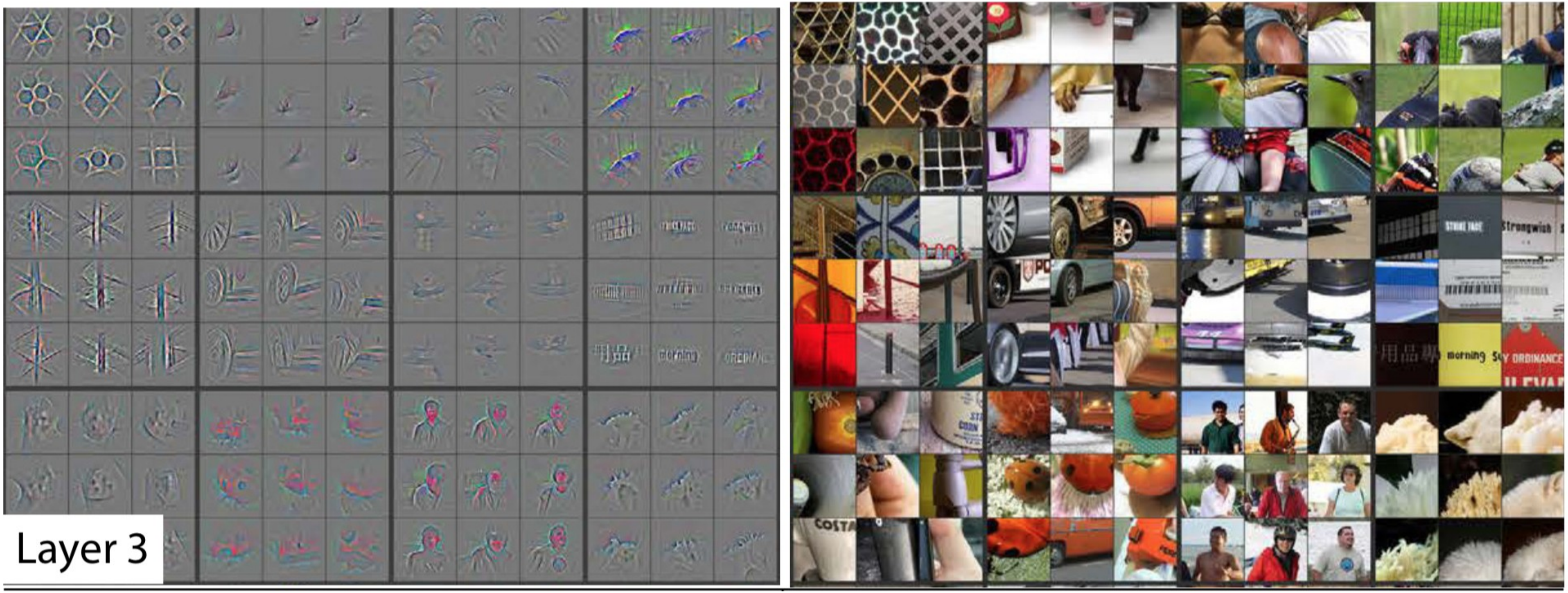


Layer 1

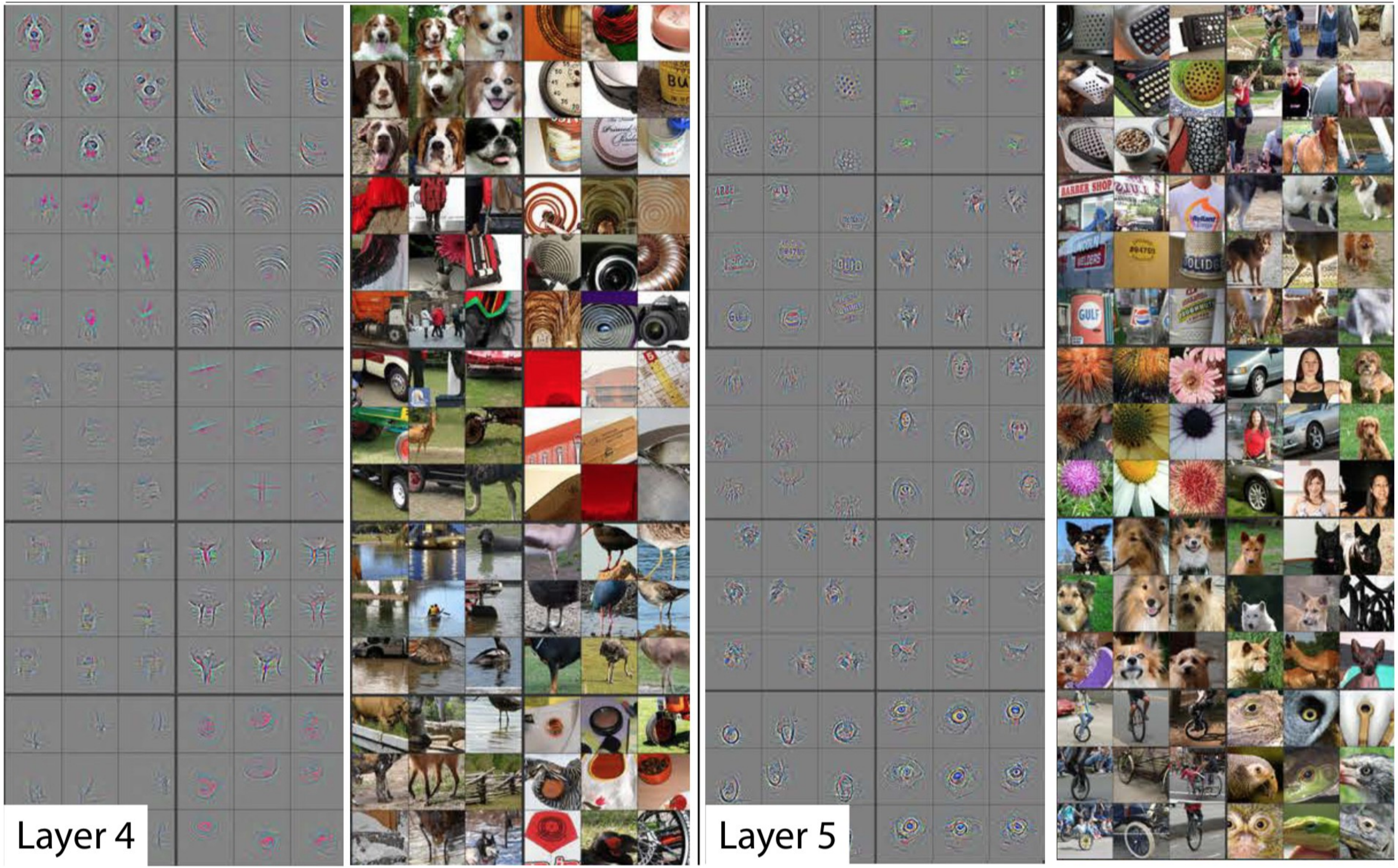




Zeiler & Fergus. Visualizing and understanding convolutional networks. *ECCV* 2014.



Zeiler & Fergus. Visualizing and understanding convolutional networks. *ECCV* 2014.



Zeiler & Fergus. Visualizing and understanding convolutional networks. *ECCV* 2014.

Pooling (downsampling) layers

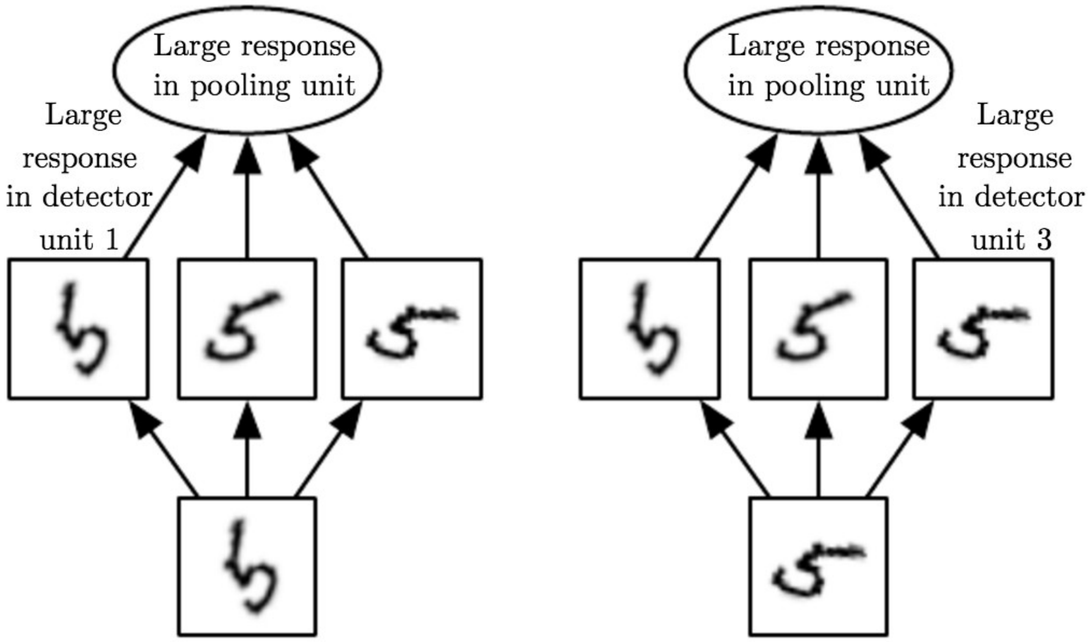


diagram: Goodfellow et al, *Deep Learning* (2016)

Variations: other activation functions

Common activation functions

- softplus:
  $f(x) = \log(1+e^x)$

- rectified linear:
  $f(x) = \max(0, x)$

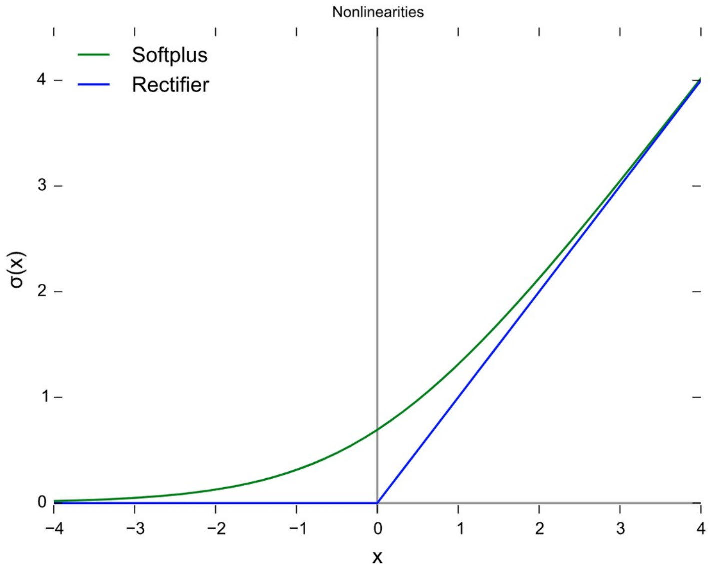
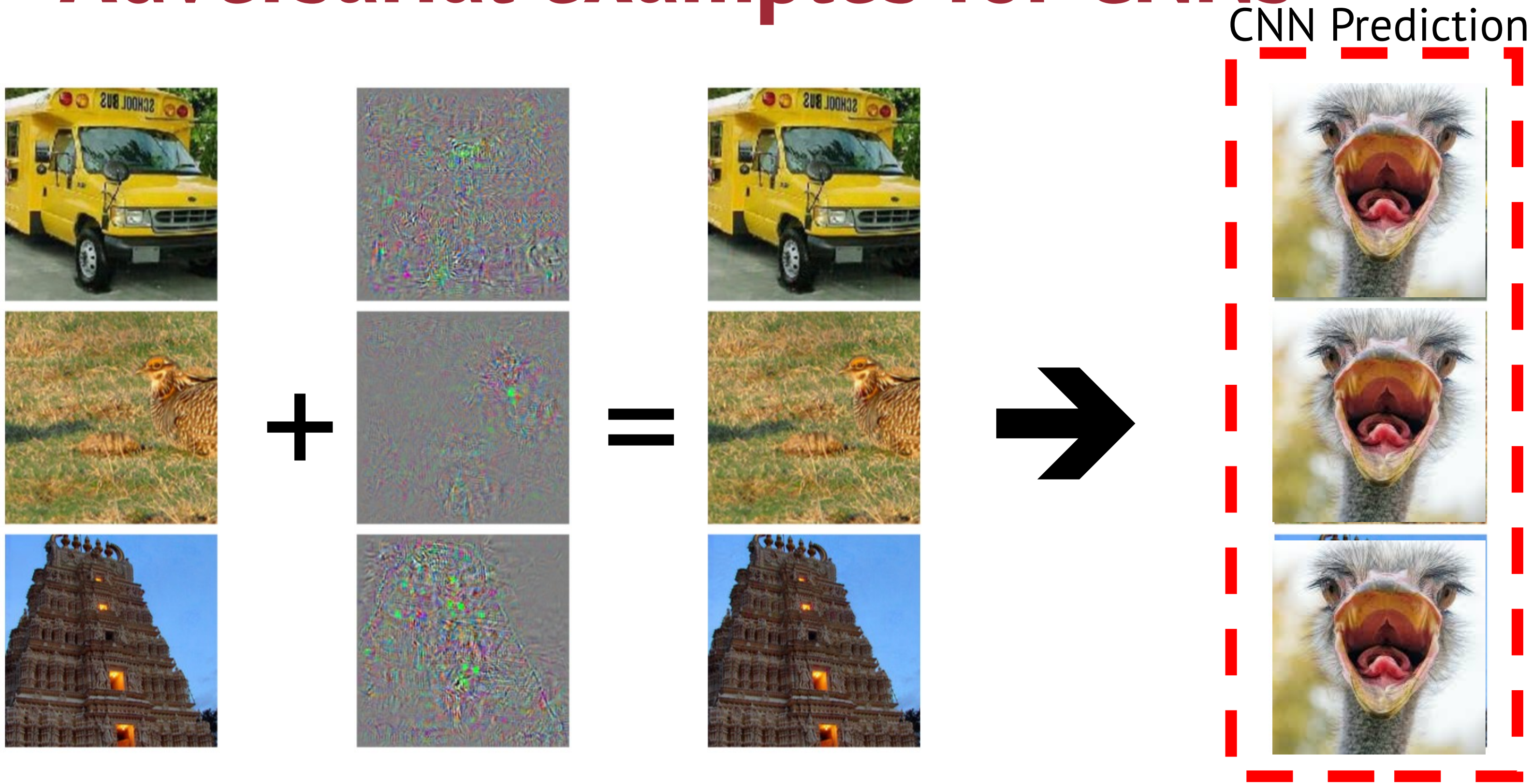


image: Stowell / Wikipedia

Adversarial examples for CNNs



Szegedy+. Intriguing properties of neural networks. *arXiv:1312.6199*, 2013.

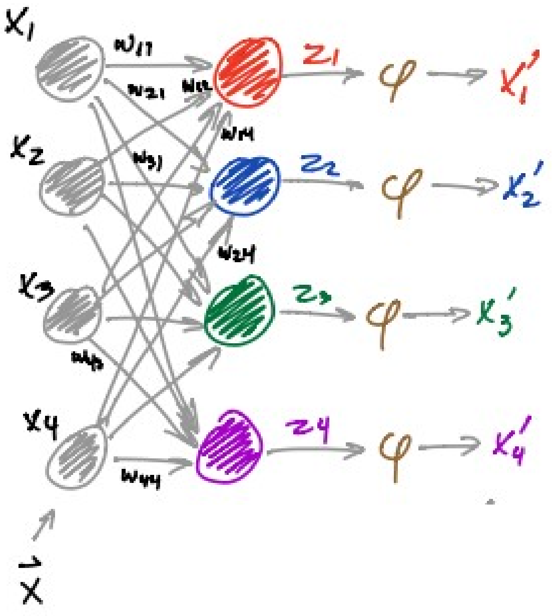
An R-duous task for CNNs

Translationally tolerant
But not rotationally tolerant

example from Geoff Hinton's talk at MIT BCS Fall Colloquium, 2014

part 3:
Computation patterns

Dense layer inference computation

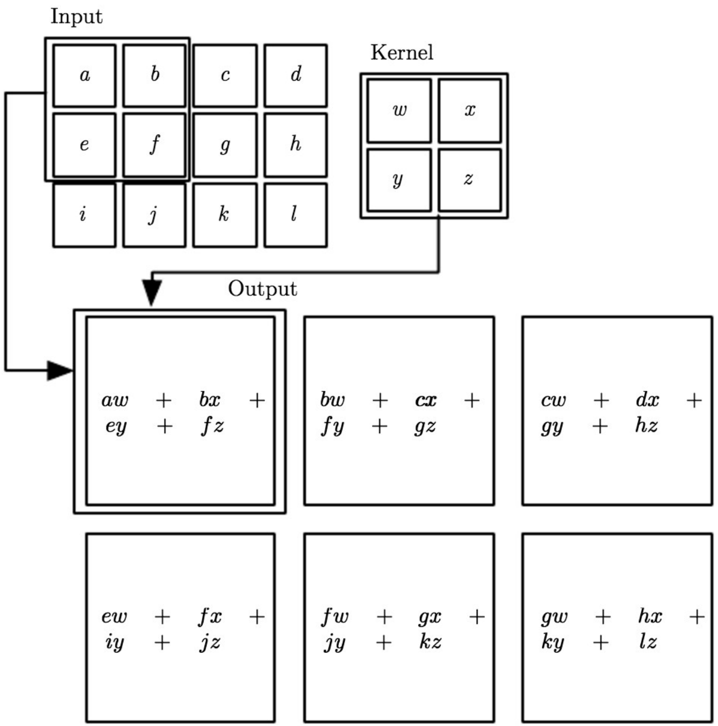


$$\vec{z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = W\vec{x} = \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} \sum_i w_{1i} x_i \\ \sum_i w_{2i} x_i \\ \sum_i w_{3i} x_i \\ \sum_i w_{4i} x_i \end{pmatrix}$$

$$\vec{x}' = \varphi(\vec{z}) = \begin{pmatrix} \varphi(z_1) \\ \varphi(z_2) \\ \varphi(z_3) \\ \varphi(z_4) \end{pmatrix}$$

can process **batches** of inputs at once → matrix × matrix (why?)

Convolutions



$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} * \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} =$$

$$aA + bB + cC + dD + eE + fF + gG + hH + iI$$

$$(I * K)(i,j) = \sum_x \sum_y I(x,y) \cdot K(i-x, j-y)$$

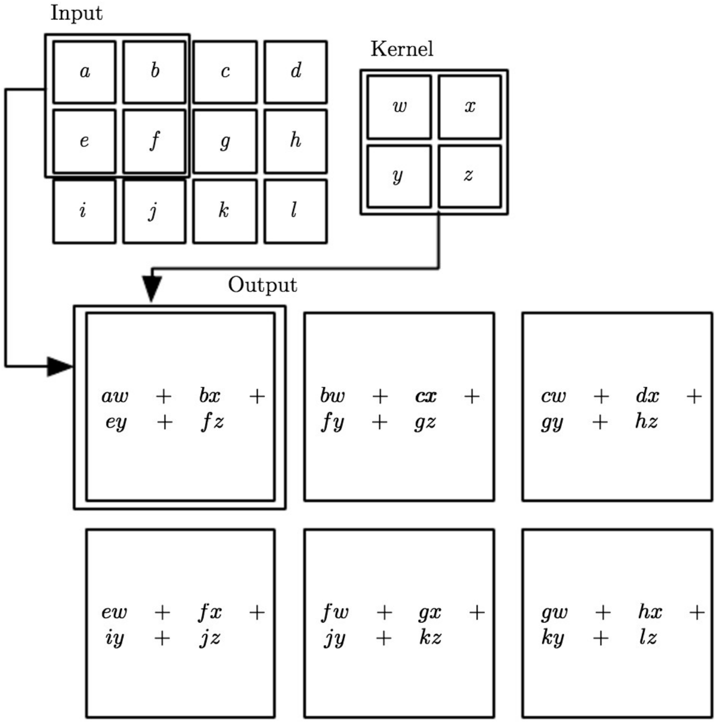
output feature map coordinates

input image coordinates

convolution filter coordinates

diagram: Goodfellow et al, *Deep Learning* (2016)

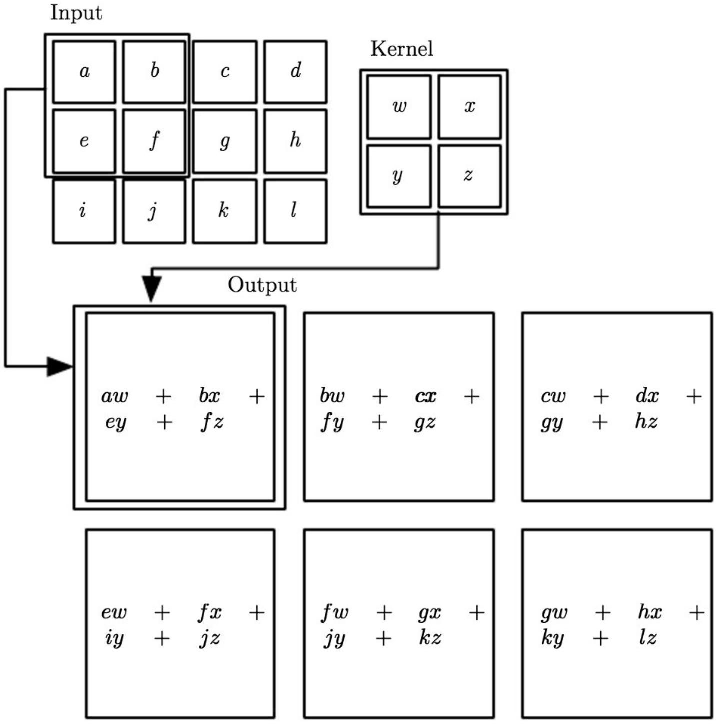
Convolutions



- edge cases
  - approach 1: ignore
  - approach 2: pad edges
    - zero, average, extend edge, ...
- stride
  - stride = 1 → typical
  - stride > 1 → a rough kind of downsampling

diagram: Goodfellow et al, *Deep Learning* (2016)

Conv. as matrix-vector multiplication



IDEA: reorganize input image

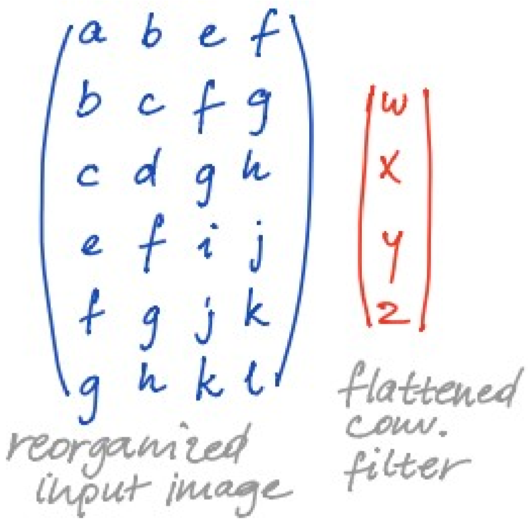
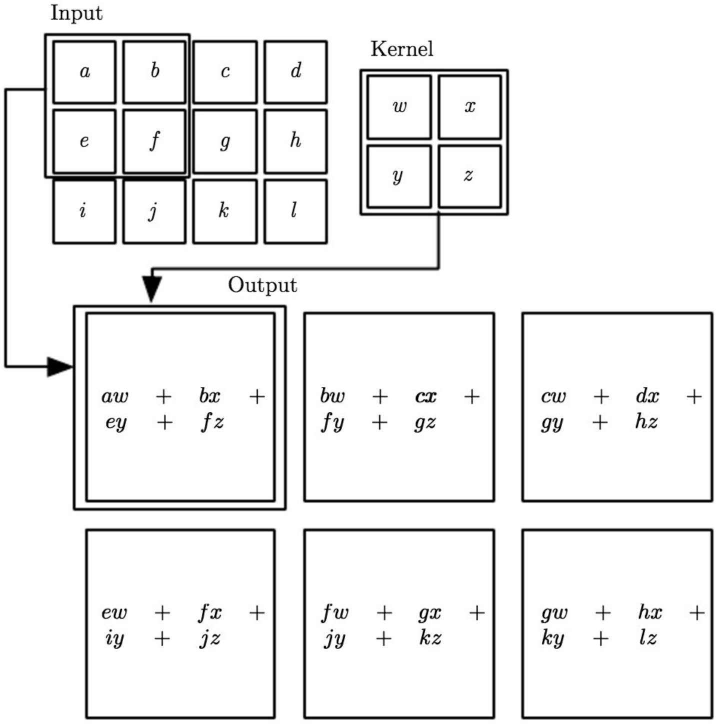


diagram: Goodfellow et al, *Deep Learning* (2016)

Convs. as matrix-matrix multiplication



IDEA: amortize across many convs.

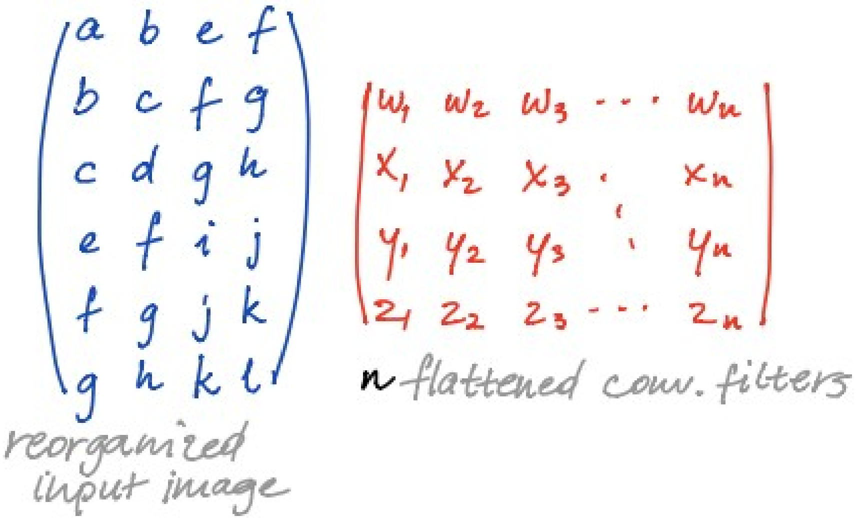
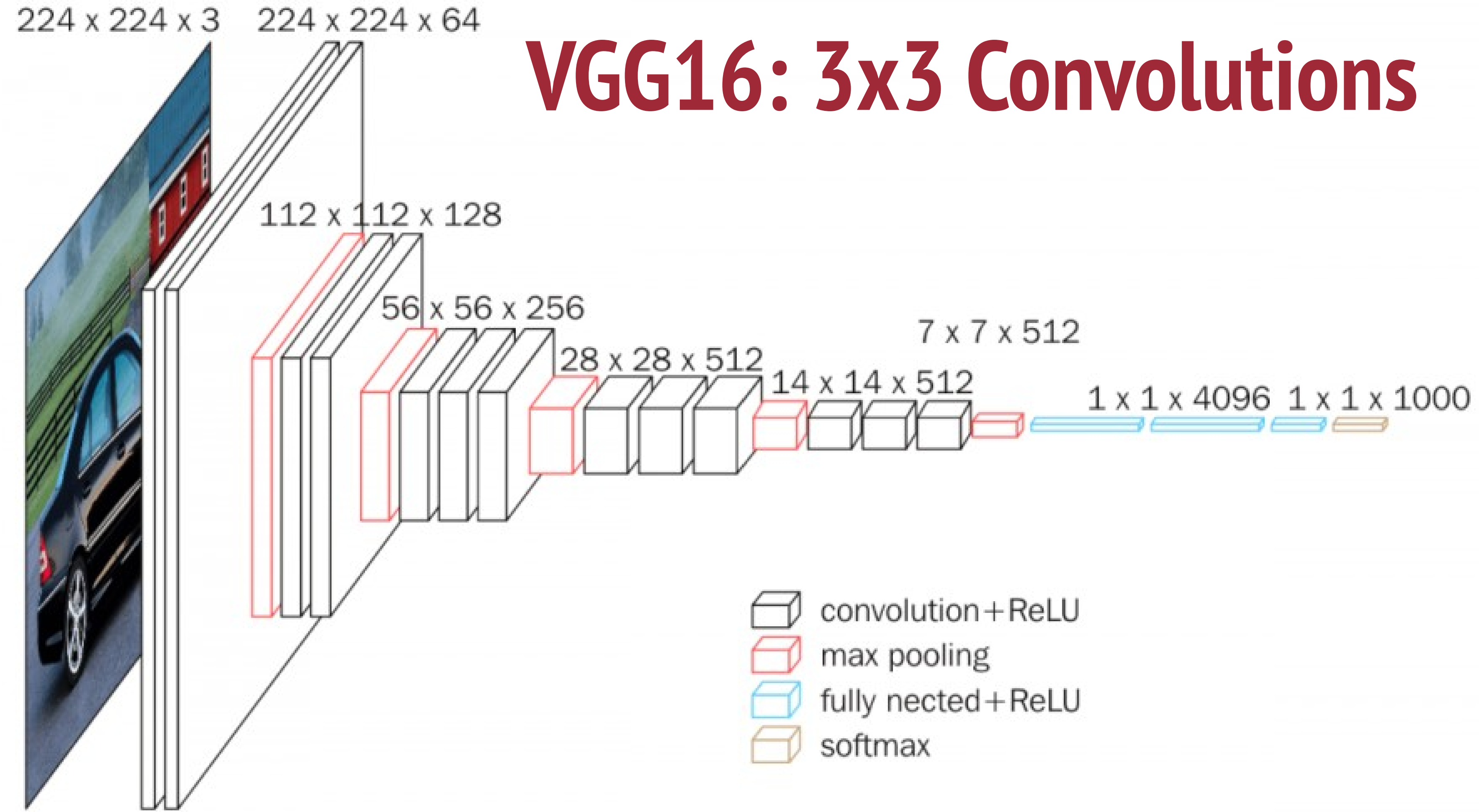


diagram: Goodfellow et al, *Deep Learning* (2016)

VGG16: 3x3 Convolutions



Simonyan & Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* 2014.

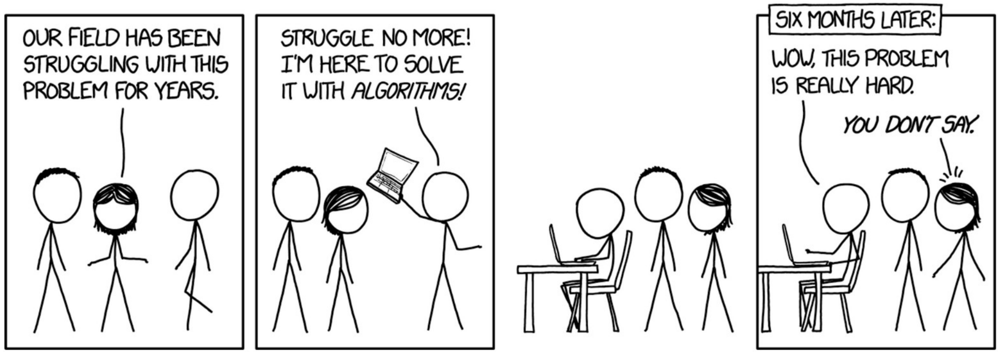
INPUT: [224x224x3] memory: 224*224*3=150K weights: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K weights: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K weights: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K weights: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K weights: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K weights: 0
FC: [1x1x4096] memory: 4096 weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 weights: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 weights: 4096*1000 = 4,096,000

TOTAL: **138M** parameters

few weights,
lots of compute,
intra-layer
reuse possible

many weights,
less compute,
reuse only across
batched inputs

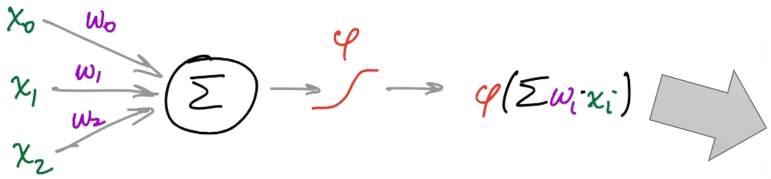
Simonyan & Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* 2014.

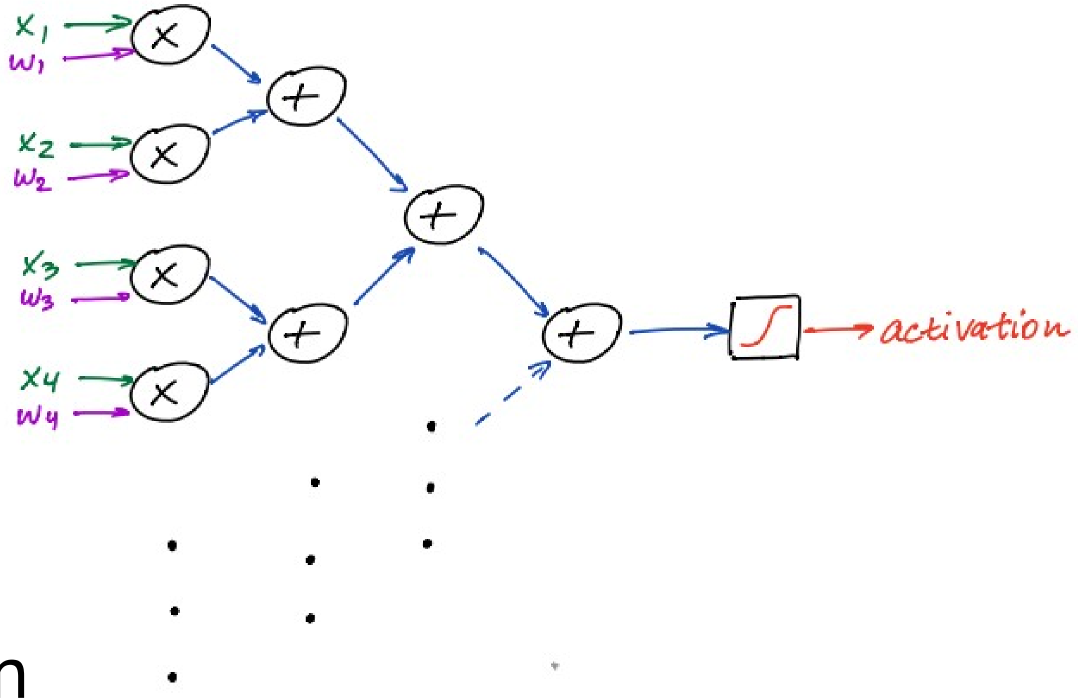


xkcd #1831

part 4: Accelerator architectures

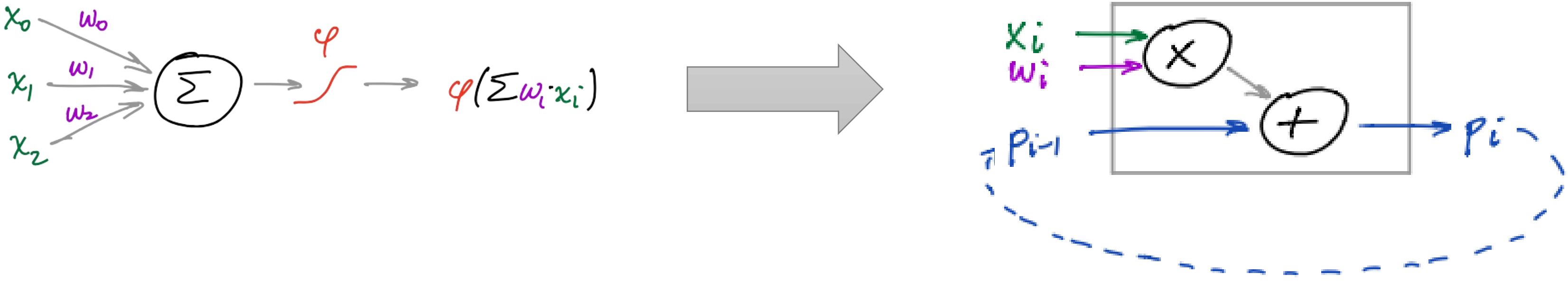
Direct neural net → hardware mapping





- ✗ generality
- ✗ scalability
- ✗ ingress bandwidth
- ✗ hardware utilization

Multiply-and-Accumulate (MAC)

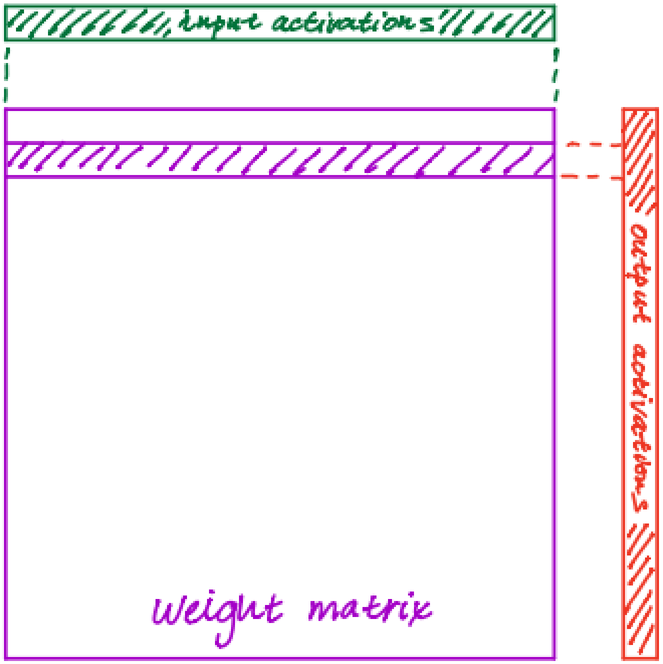


- sum can be broken down into multiple multiply-and-accumulate (MAC) operations

- partial sum *p* fed back to the adder, eventually produces final result

Computing one FC layer

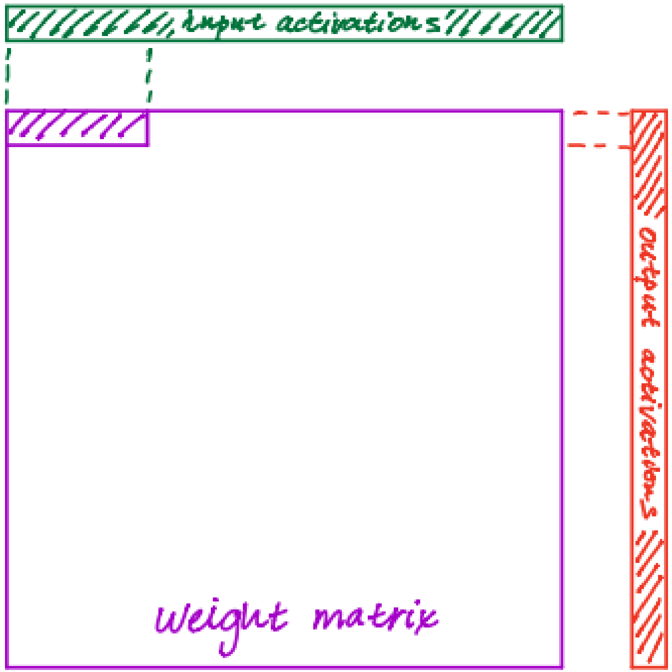
```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    for i in [0..N_in):
        sum[n] += w[n][i] * in[i]
    out[n] = φ(sum[n])
```



problem: no parallelism

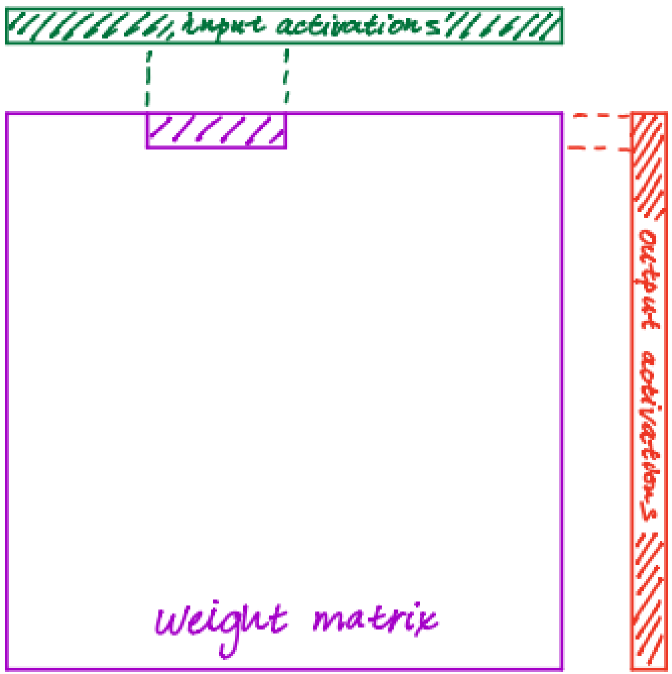
Parallelism across HW

```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    psum[0..N_HW) = 0
    for ii in [0..N_in/N_HW):
        parfor i in [0..N_HW):
            psum[i] += w[n][i] * in[ii*N_HW+i]
    out[n] = φ(sum(psum[0..N_HW)))
```



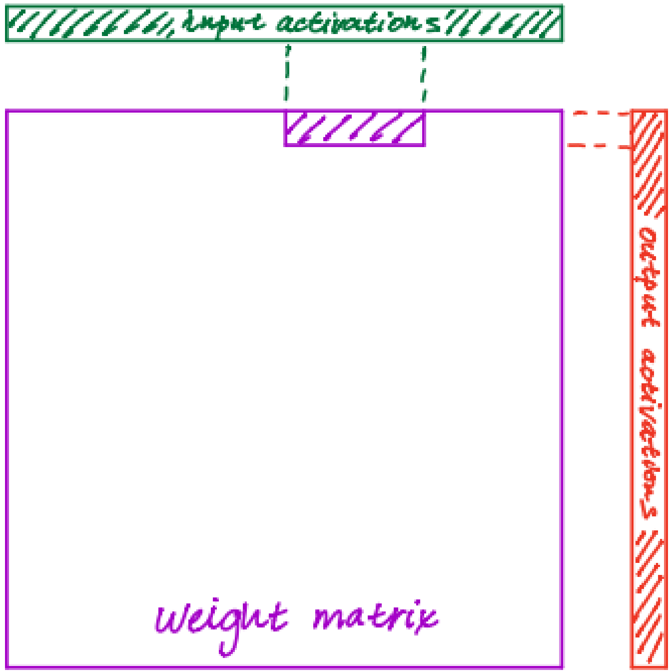
Parallelism across HW

```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    psum[0..N_HW) = 0
    for ii in [0..N_in/N_HW):
        parfor i in [0..N_HW):
            psum[i] += w[n][i] * in[ii*N_HW+i]
    out[n] = φ(sum(psum[0..N_HW)))
```



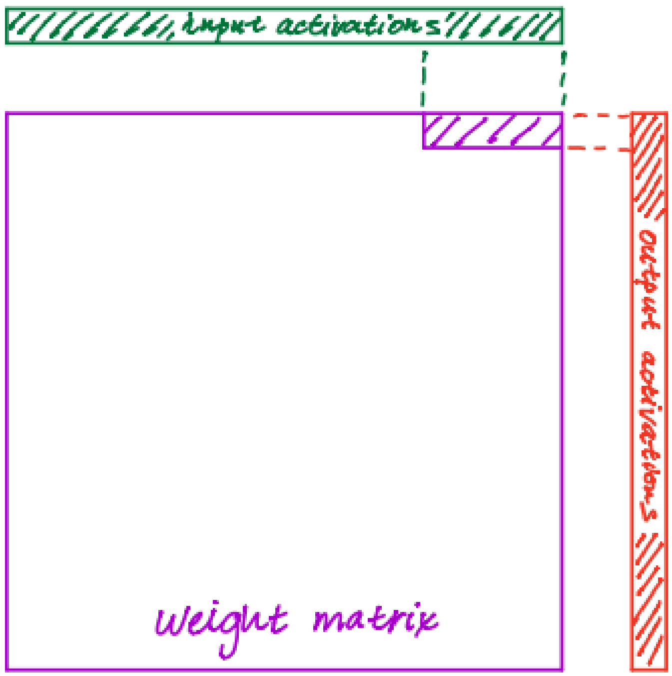
Parallelism across HW

```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    psum[0..N_HW) = 0
    for ii in [0..N_in/N_HW):
        parfor i in [0..N_HW):
            psum[i] += w[n][i] * in[ii*N_HW+i]
    out[n] = φ(sum(psum[0..N_HW)))
```



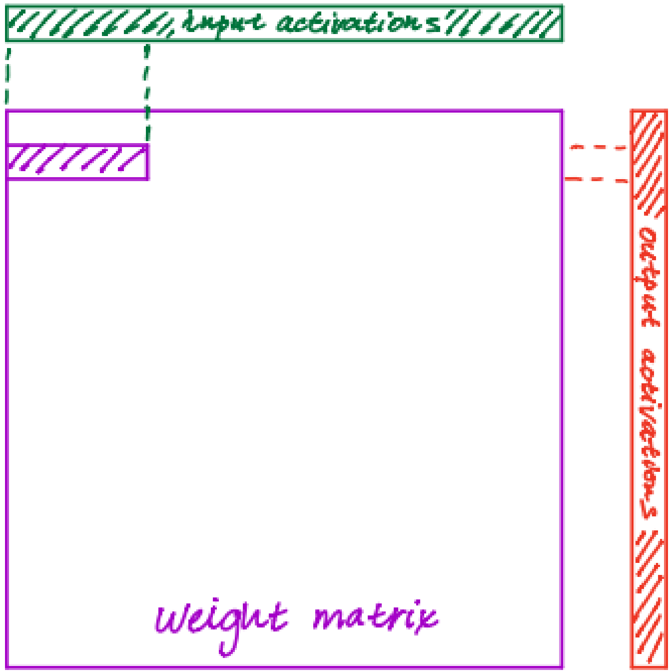
Parallelism across HW

for n in [0..N_{out}):
    sum[n] = 0
for n in [0..N_{out}):
    psum[0..N_{HW}) = 0
    for ii in [0..N_{in}/N_{HW}):
        parfor i in [0..N_{HW}):
            psum[i] += w[n][i] * in[ii*N_{HW}+i]
    out[n] = φ(sum(psum[0..N_{HW})))



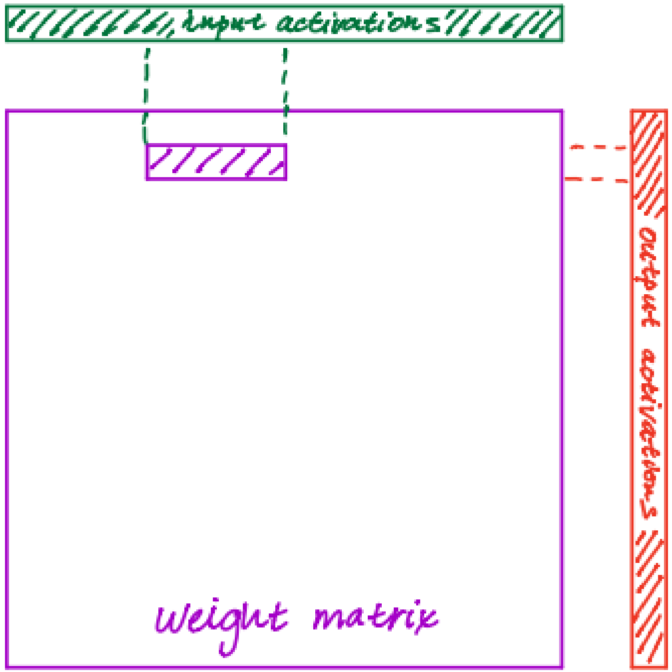
Parallelism across HW

```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    psum[0..N_HW) = 0
    for ii in [0..N_in/N_HW):
        parfor i in [0..N_HW):
            psum[i] += w[n][i] * in[ii*N_HW+i]
    out[n] = φ(sum(psum[0..N_HW)))
```



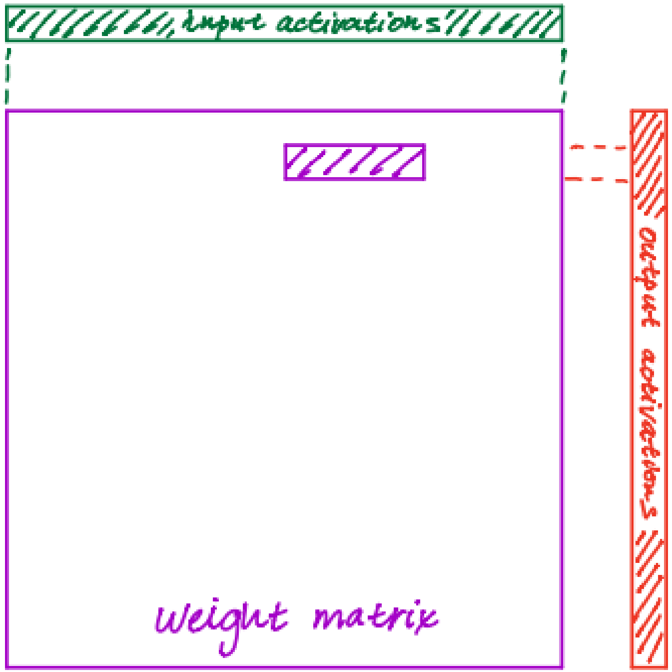
Parallelism across HW

```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    psum[0..N_HW) = 0
    for ii in [0..N_in/N_HW):
        parfor i in [0..N_HW):
            psum[i] += w[n][i] * in[ii*N_HW+i]
    out[n] = φ(sum(psum[0..N_HW)))
```



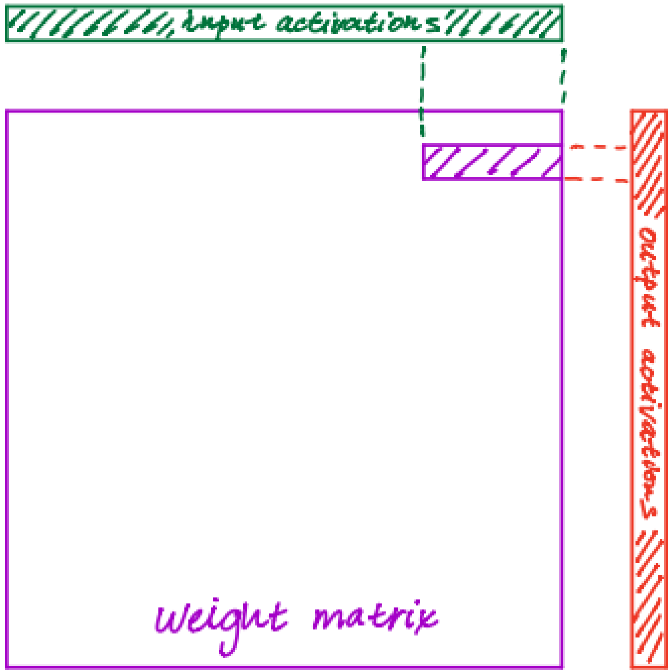
Parallelism across HW

```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    psum[0..N_HW) = 0
    for ii in [0..N_in/N_HW):
        parfor i in [0..N_HW):
            psum[i] += w[n][i] * in[ii*N_HW+i]
    out[n] = φ(sum(psum[0..N_HW)))
```



Parallelism across HW

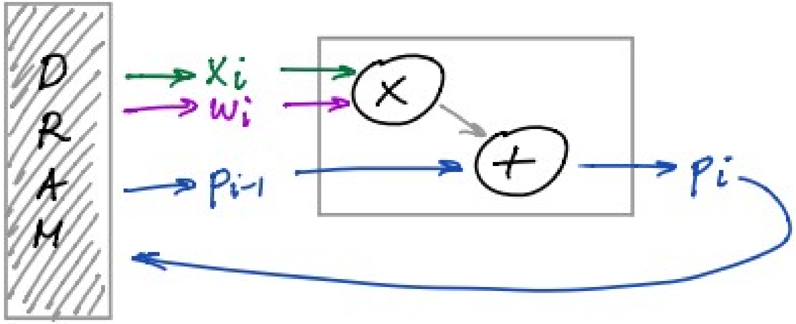
```
for n in [0..N_out):
    sum[n] = 0
for n in [0..N_out):
    psum[0..N_HW) = 0
    for ii in [0..N_in/N_HW):
        parfor i in [0..N_HW):
            psum[i] += w[n][i] * in[ii*N_HW+i]
    out[n] = φ(sum(psum[0..N_HW)))
```



problem: each input fetched many times

Multiply-and-Accumulate (MAC)

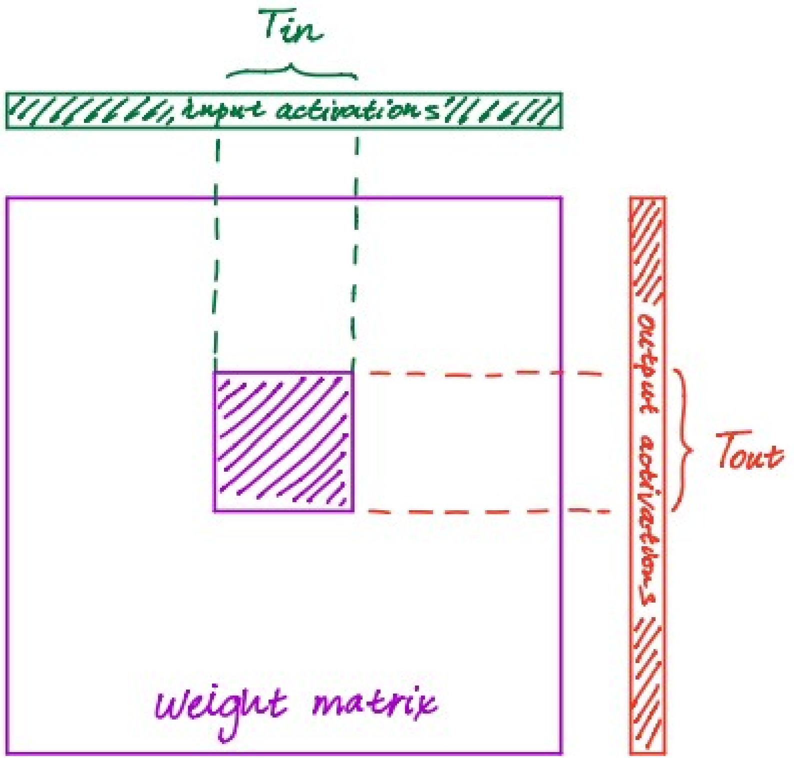
- worst case:
  - inputs, wts ← DRAM
  - outputs → DRAM
  - partial sums ←→ DRAM



- low bandwidth, accesses cost ~100× more energy
- **IDEA:** use on-chip storage to reuse input activations

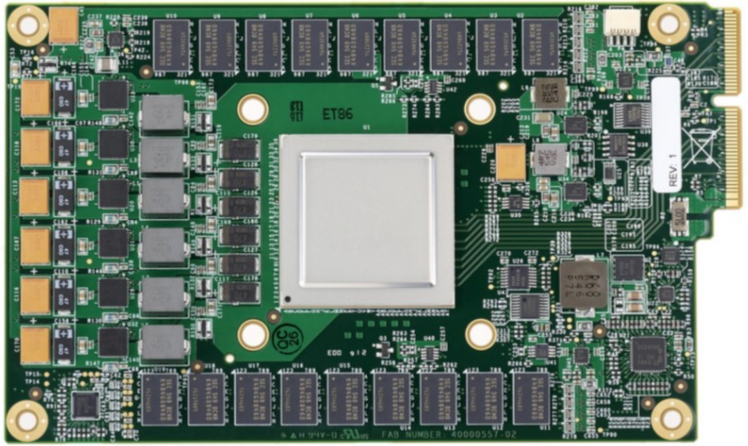
Tile-based processing

- Reuse T_{in} inputs for each of T_{out} outputs
  - where $T_{in} \times T_{out} = N_{HW}$
  - need extra psum storage



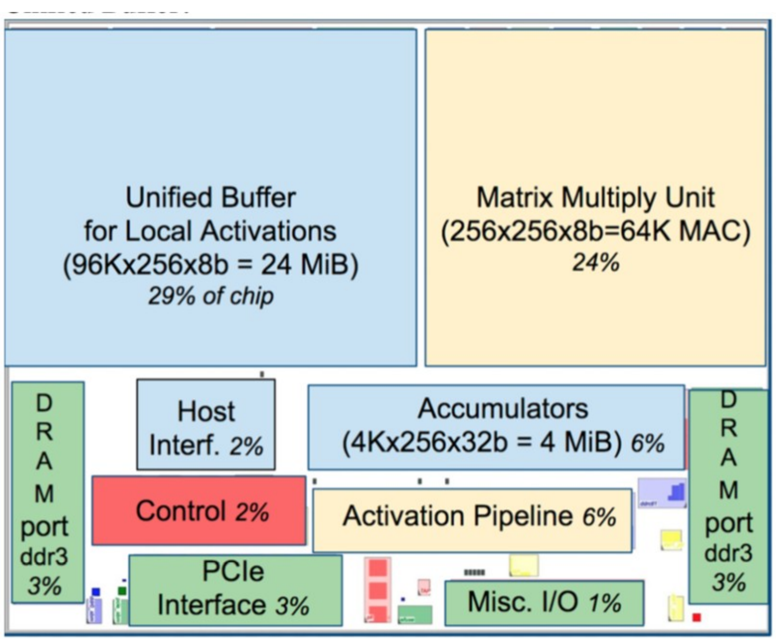
The Google TPU

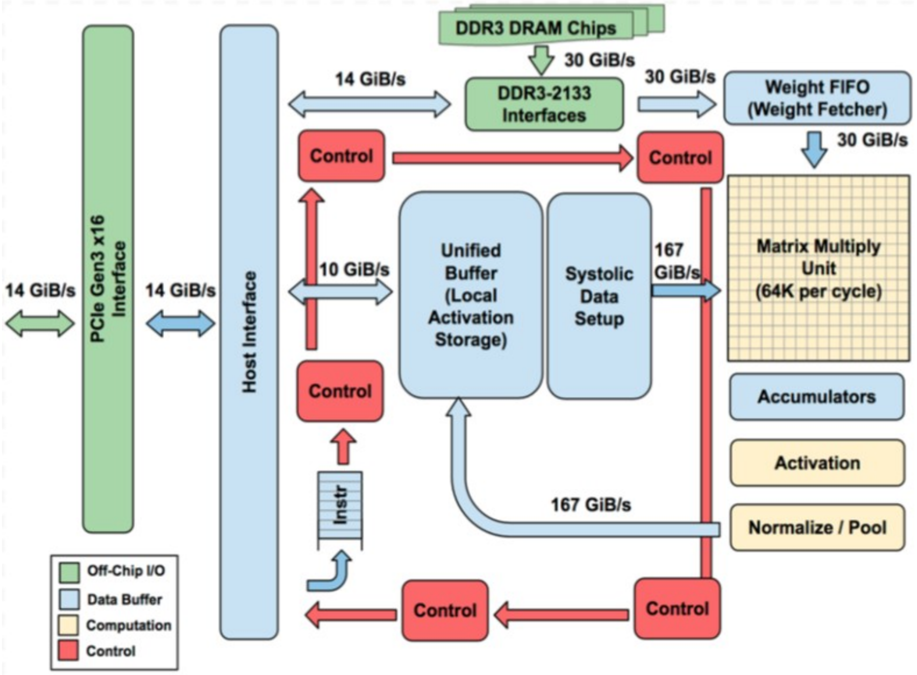
- workload characteristics
  - process batches of inputs
  - mostly MLPs and LSTMs – only 5% workload are CNNs
  - user-facing apps, must have predictable latency
- **matrix multiply accelerator**: 256×256 8-bit MACs
- CISC (!)



Jouppi+. In-Datacenter Performance Analysis of a Tensor Processing Unit. *ISCA* 2017.

The Google TPU

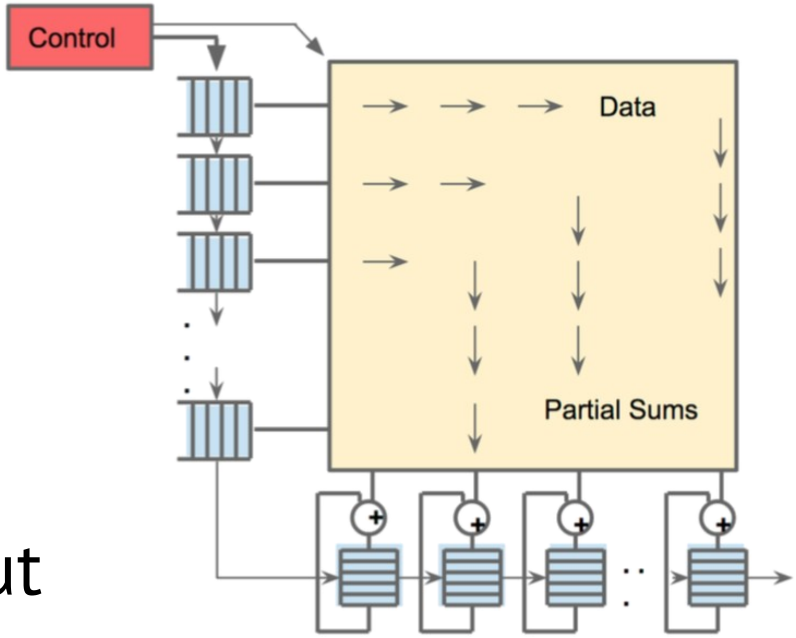




Jouppi+. In-Datacenter Performance Analysis of a Tensor Processing Unit. *ISCA* 2017.

The matrix multiply unit: a systolic array

- weights loaded at init
- inputs fed from one side
  - each processing element propagates input and partial sum
- sums fall out the bottom
- **KEY IDEA**: values flow without RF access for intermediates



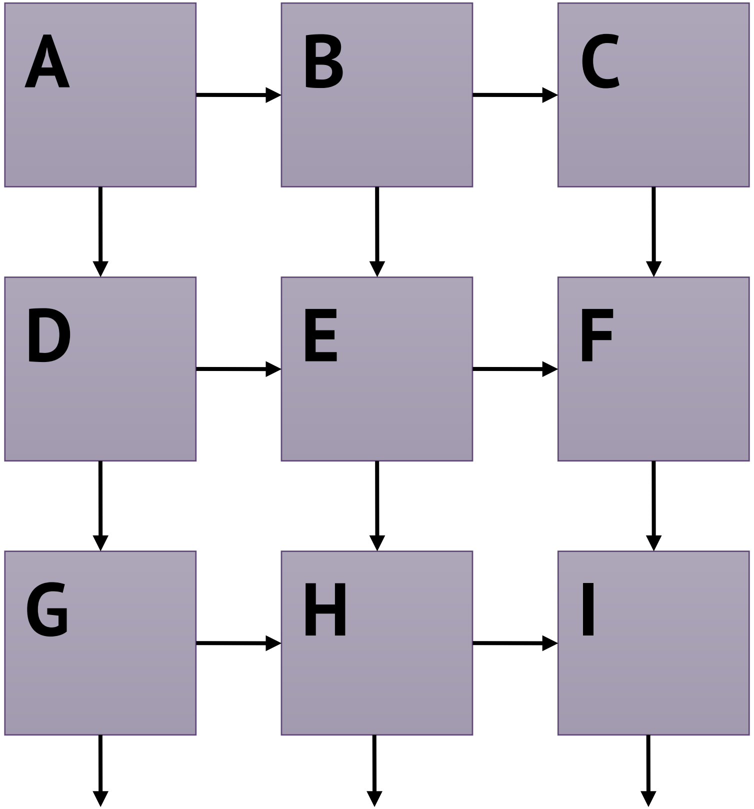
Jouppi+. In-Datacenter Performance Analysis of a Tensor Processing Unit. *ISCA* 2017.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} \quad & \quad & \quad \\ & & \\ & & \end{pmatrix}$$

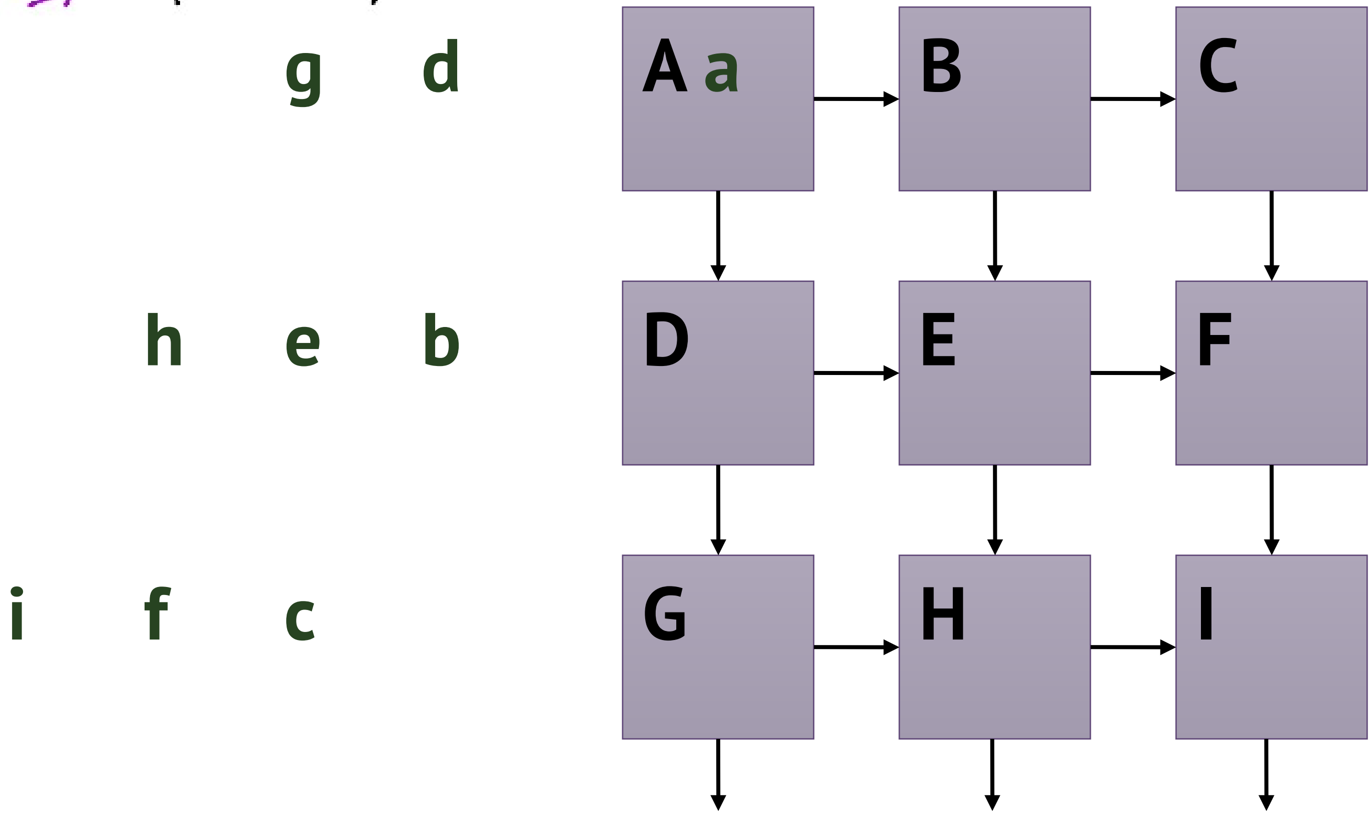
g d a

h e b

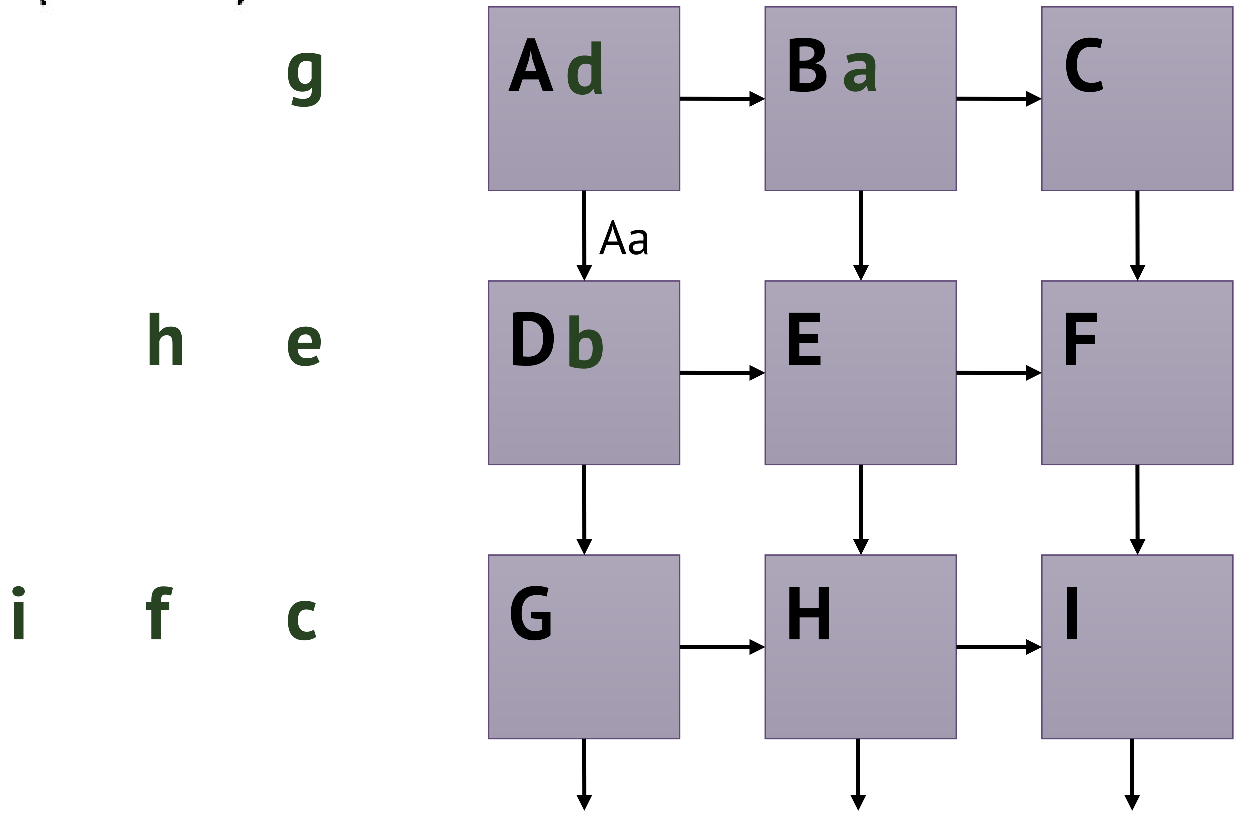
i f c



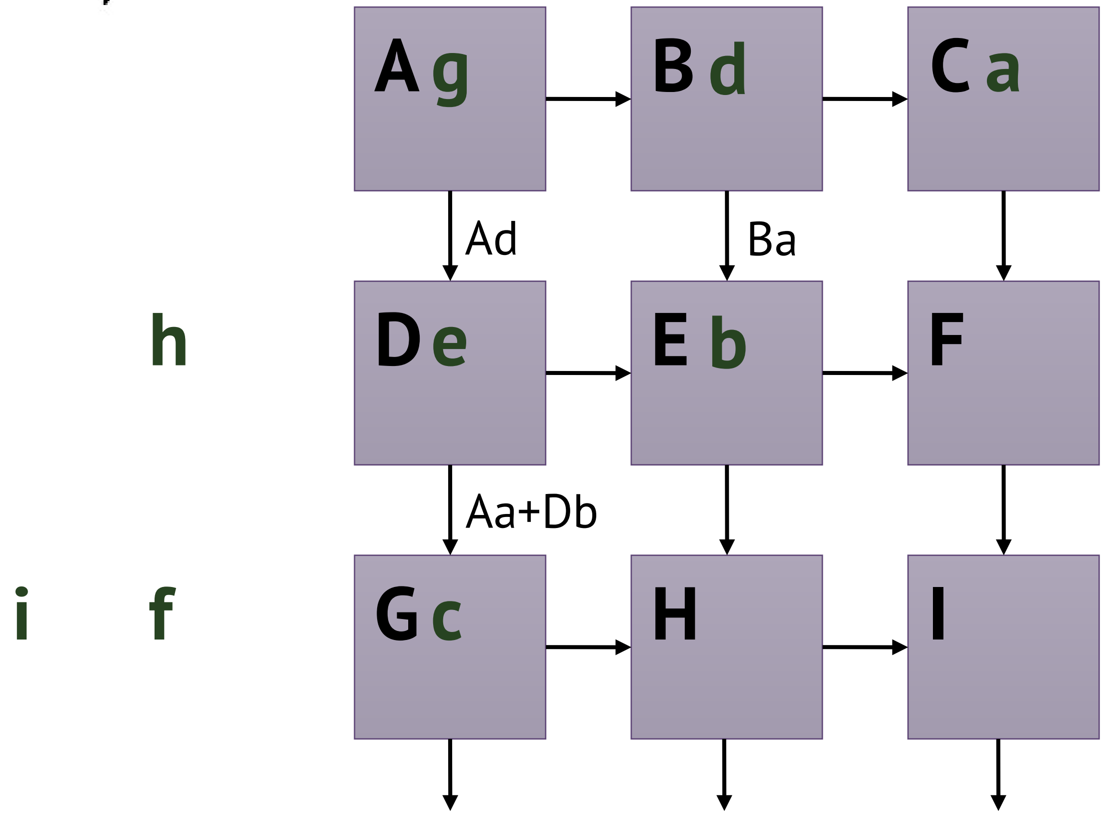
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix}$$



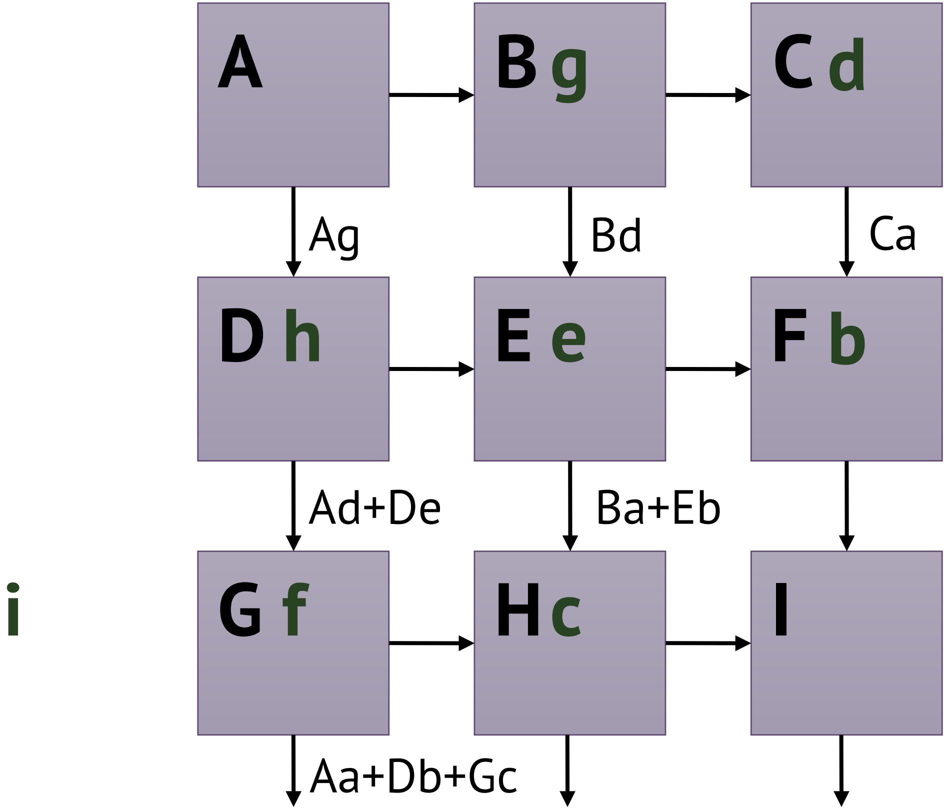
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix}$$



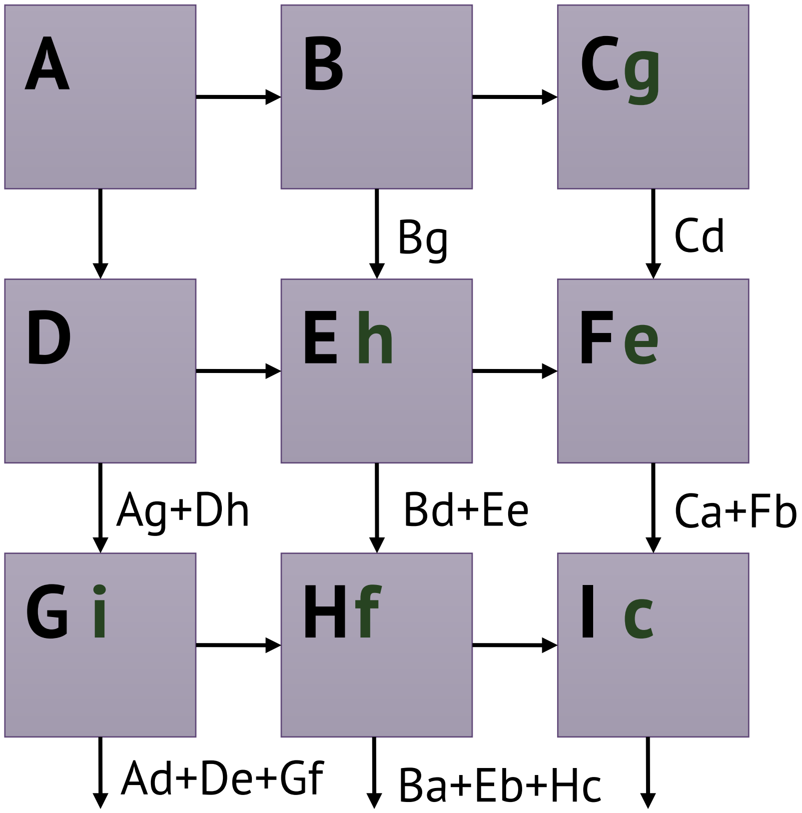
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} & & \\ & & \\ & & \end{pmatrix}$$



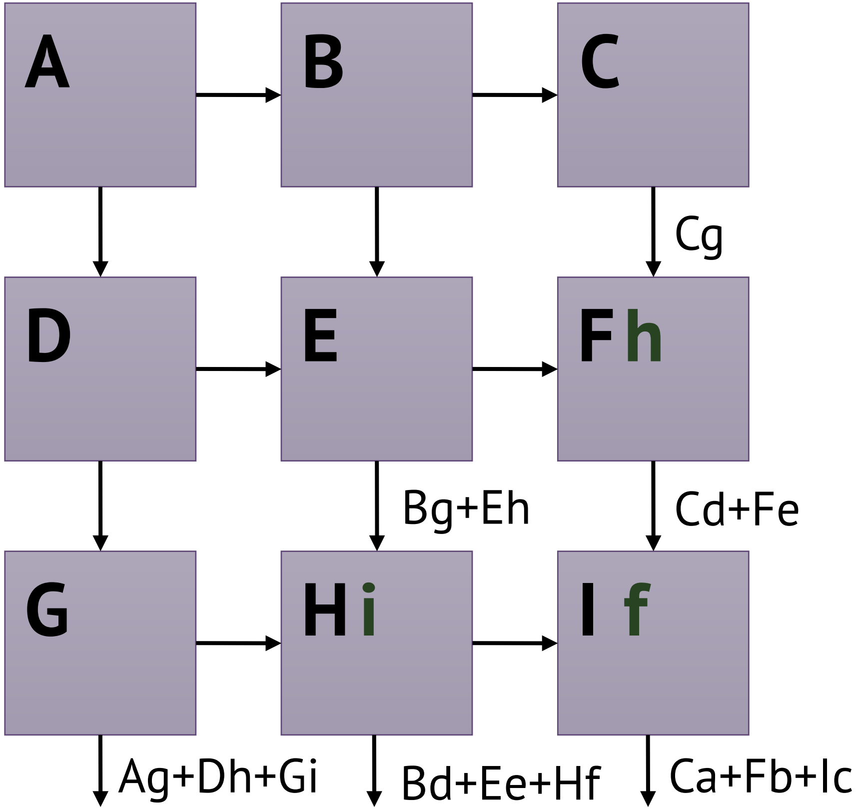
$$\begin{pmatrix}a&b&c\\d&e&f\\g&h&i\end{pmatrix}\begin{pmatrix}A&B&C\\D&E&F\\G&H&I\end{pmatrix}=\begin{pmatrix}\checkmark&&\\&&\\&&\end{pmatrix}$$



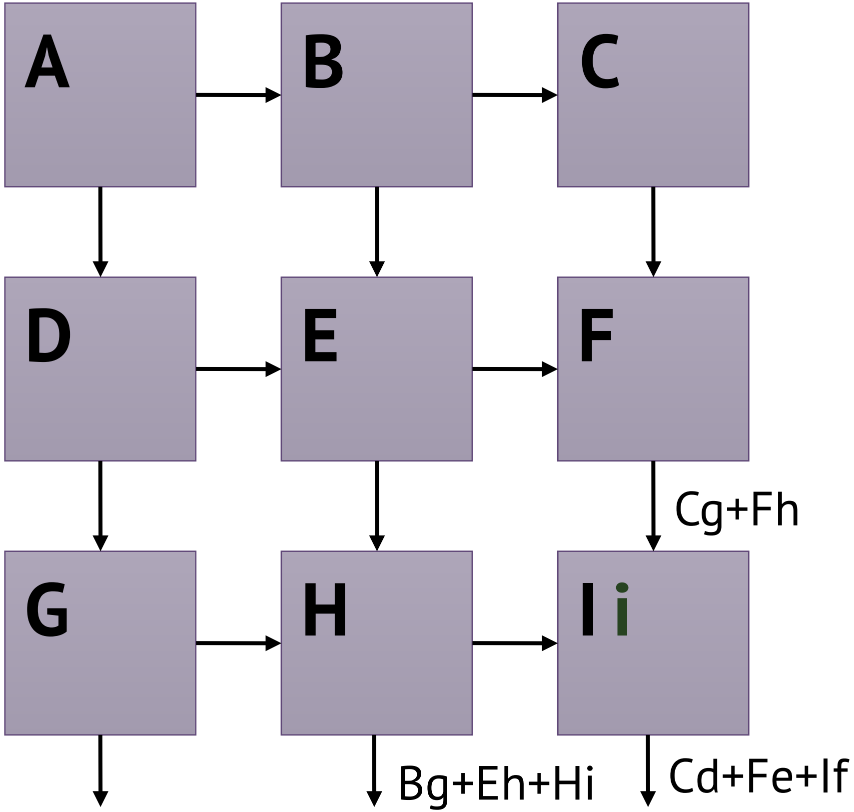
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} \checkmark & \checkmark & \\ \checkmark & & \\ & & \end{pmatrix}$$



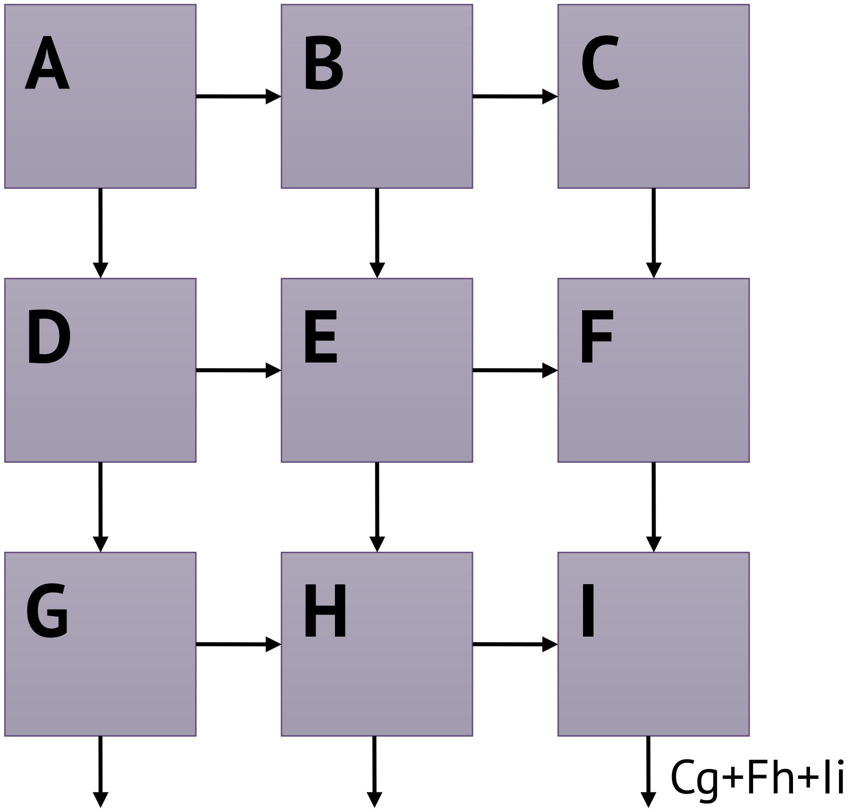
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \\ \checkmark & & \end{pmatrix}$$



$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \end{pmatrix}$$



$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \end{pmatrix}$$



$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} = \begin{pmatrix} \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \\ \checkmark & \checkmark & \checkmark \end{pmatrix}$$